EXPERT'S VOICE IN OPEN SOURCE

Architecting, Developing,
and Administering MongoDB

MongoDB实战

架构、开发与管理

[美] Shakuntala Gupta Edward 著
Navin Sabharwal
蒲成 译

Apress®

清华大学出版社

MongoDB 实战

架构、开发与管理

[美] Shakuntala Gupta Edward 著
Navin Sabharwal
蒲 成 译

清华大学出版社
北 京

Shakuntala Gupta Edward, Navin Sabharwal
Practical MongoDB: Architecting, Developing, and Administering MongoDB
EISBN：978-1-4842-0648-5

北京市版权局著作权合同登记号　图字：01-2016-8570

图书在版编目(CIP)数据

MongoDB 实战　架构、开发与管理 /(美) 夏琨塔拉 • 古普塔 • 爱德华 等著；蒲成译. —北京：清华大学出版社，2017
书名原文：Practical MongoDB:Architecting, Developing,and Administering MongoDB
ISBN 978-7-302-45673-5

Ⅰ. ①M… Ⅱ. ①夏… ②蒲… Ⅲ. ①关系数据库系统 Ⅳ. ①TP311.138

中国版本图书馆 CIP 数据核字(2016)第 281183 号

责任编辑：王　军　于　平
装帧设计：牛静敏
责任校对：成凤进
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京鑫丰华彩印有限公司
装 订 者：三河市溧源装订厂
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：16.25　　**字　　数**：375 千字
版　　次：2017 年 1 月第 1 版　　**印　　次**：2017 年 1 月第 1 次印刷
印　　数：1~3500
定　　价：49.80 元

产品编号：071606-01

译 者 序

随着以用户为中心的社交类网站的兴起，尤其是在当前飞速发展的移动互联时代背景下，传统的关系数据库在应对海量的信息，特别是超大规模和高并发的社交网络类型应用所带来的爆炸式数据时已经显得力不从心，暴露了很多难以克服的问题，而非关系型数据库则由于其本身的特点能够快速适应这些应用场景，因而得到了非常迅速的发展。

虽然 NoSQL 的流行不过短短数年时间，但不可否认的是，在持续不断的版本迭代之下，现在的 NoSQL 系统已经更加成熟、稳定。在这众多的 NoSQL 工具中，MongoDB 可以说是其中的佼佼者，它可以为大数据建立快速、可扩展的存储库，从而满足日新月异的应用场景需求。

MongoDB 是一个基于分布式文件存储的数据库。它的基础开发语言是 C++。其目的在于为基于互联网的应用提供可扩展的高性能数据存储解决方案。对于大型互联网公司以及处在飞速发展之中的互联网公司来说，以 MongoDB 为代表的 NoSQL 数据库产品正逐渐成为其无法绕过的必备部署工具之一。

实际上，MongoDB 是一个介于关系型数据库和非关系型数据库之间的产品，它是非关系型数据库中功能最丰富、最像关系型数据库的一个。它支持松散的数据结构，即基于 JSON 的 BSON 格式，因而在实际使用中，MongoDB 可以存储较为复杂的数据类型。它的特点是高性能、易部署、易使用，并且存储数据非常方便。不过，其最大的优势在于所支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系型数据库单表查询的绝大部分功能，此外还支持对数据建立索引。

本书的两位作者都是数据库和大数据应用领域经验丰富的实践专家，他们在数据分析应用领域的见解自然是独一无二且领先于行业的。相信在阅读完本书并且充分吸收书中的理念之后，读者将对 MongoDB 有一个全面了解，从而帮助读者深刻理解 MongoDB 在现实环境中的应用与实践。本书使用了大量贴合实际的示例，结合作者的亲身实践以及多年的丰富经验来介绍 MongoDB 的概念与应用原则。重点在于让读者理解其价值和实现的机制及原理，而不是空谈枯燥乏味的理论知识点。

本书提供了使用 MongoDB 平台进行架构、开发与部署应用程序的清晰指导与实践示例。数据库开发人员、架构师和管理员将在本书中找到涵盖 MongoDB 平台所有知识点的有用信息，以及如何将它用于实践的内容。当下，互联网、特别是移动互联网正处于高速发展时期，已有的工具要跟上时代的步伐就必须不断地推陈出新，MongoDB 的开发团队也在不知疲倦地对 MongoDB 进行更新和调整，使其能跟上时代发展的脚步。作

为一本 MongoDB 实践领域的权威书籍，本书能让你快速、专业地掌握以 MongoDB 为基础的重点知识，从而构建高效、高并发的应用。

在此要特别感谢清华大学出版社的编辑们，在本书翻译过程中他们提供了颇有助益的帮助，没有其热情付出，本书将难以付梓。

本书全部章节由蒲成翻译，参与本次翻译的还有何东武、李鹏、林超、刘洋洋、茆永锋、潘丽臣、王滨、申成龙、王佳、负书谦。由于译者水平有限，难免会出现一些错误或翻译不准确的地方，如果有读者能够指出并勘正，译者将不胜感激。

译 者

作者简介

Shakuntala Gupta Edward 从 10 年前就开始使用数据库技术。她的经验涵盖了 SQL Server、Oracle 数据库、Analytics 平台以及大数据技术，例如 MongoDB、Cassandra 和 SAP HANA。

Shakuntala 是一位才华横溢的架构师，擅长于利用各种数据库技术为各种业务领域构建产品和解决方案。

Shakuntala 一直在参与利用大数据技术 MongoDB 和 Cassandra 来开发产品和解决方案的工作。Shakuntala 拥有计算机应用的硕士学位。

Navin Sabharwal 是一位创新者、思想领袖、作者、顾问。他专注于报表与分析领域，包括 SQL Server、Oracle、MySql 在内的 RDBMS 技术以及包括 Hadoop、MongoDB 和 SAP HANA 在内的大数据技术。Navin 一直在使用大数据技术为 IT 服务管理、产品开发、云计算、云生命周期管理以及社交网络产品开发领域构建产品和服务。

Navin 已经构建出有良好商机的屡获殊荣的产品和解决方案，并且在各个领域都取得了大量专利，比如 IT 服务、评估引擎、排名算法、容量规划引擎以及知识管理。

Navin 还著有以下书籍：*Cloud Computing First Steps*(CreateSpace 出版，ISBN#: 978-1478130086)、*Apache Cloudstack Cloud Computing*(Packt Publishing 出版，ISBN#: 978-1782160106)、*Cloud Capacity Management*(Apress 出版，ISBN #: 978-1430249238)。Navin 拥有信息技术方面的硕士学位，并且是经认证的项目管理专家(Certified Project Management Professional)。

技术审校者简介

Prasoon Kumar 是一位经验丰富的技术专家和培训师，他在构建世界一流的软件产品方面拥有超过 18 年的丰富经验。他在 1993 年的印度工程学系入学考试(IIT-JEE)中从 10 万名考生中脱颖而出，该入学考试可以说是世界上最具挑战性和竞争性的考试。

他现在主要居住在班加罗尔，与印度和美国硅谷的像 MongoDB、Justdial、Avaya 等这样的公司都有广泛接触。他使用 MySQL、MongoDB、HBase、Apache Solr、Elasticsearch、PHP、Node.JS 来为复杂应用程序构建可扩展的后端。

他担当推广用于横向扩展数据存储的混合持久化以及 NoSQL 解决方案的角色。他帮助印度的大型电子商务、FSI、医疗卫生和出版公司解决了文档管理、高流量网站的扩展性需求问题。他已经为 TB 量级的数据存储做过调整、优化、备份、恢复、迁移和升级。他在 http://prasoonk.wordpress.com 上发表了关于技术、商业、黑客马拉松以及创业精神的博文。

Sundar Rajan Raman 是一位大数据架构师，他目前供职于美国银行。他拥有位于印度锡尔杰尔的国立技术学院的技术学士学位。在他供职于 AT&T、新加坡电信、德意志银行期间，他逐渐成为经验丰富的 Java 和 J2EE 编程人员。他是一位消息传递平台的专家，在 Sonic MQ、Websphere MQ、TIBCO 方面拥有丰富的经验，并且具有这些产品各自的认证。他当前主要专注于大数据技术。目前他正在使用 Hadoop 及其生态系统，比如 Pig、HIVE、Oozie 和 Storm、Spark 等。他为 AT&T 架构了一个基于 MongoDB 的分析引擎。

致　谢

要特别感谢那些帮助编著本书的人，Rajeev Pratap Singh 和 Amit Agrawal 为书中的代码片段提供了帮助，Dheeraj Raghav 为本书的内容设计提供了创意。

万分感谢 Stuti Awasthi，他促成了本书并且为本书提供了灵感。

本书作者要感谢大数据技术和开源社区的创建者，因为他们提供了如此强大的工具和技术用于编码，并且使轻易快速解决真实业务问题的产品和解决方案得以实现。

前　言

如今，数据仓库作为一个行业已经存在很多年了。关系型数据库被用于存储数据已经几十年了，同时 SQL 已经成为实际上的与 RDBMS 交互的语言。随着社交网络、物联网以及互联网上巨量的非结构化数据的涌现，数据存储、处理以及分析的需求正逐渐爆发。传统的 RDBMS 系统和存储技术并非旨在处理各种各样海量的数据。

因此，大数据技术诞生了，如今它推动着各个互联网规模公司及其巨量数据的发展。像 Facebook、Twitter、Google 以及雅虎这样的公司正在利用大数据技术提供互联网规模的产品和服务，它们能够支持数百万的用户。

本书将帮助读者理解大数据技术、其出现的背景、需求，然后我们将介绍与使用 MongoDB 架构解决方案有关的深层技术观点。本书将让读者能够理解适合使用大数据技术的关键用例，也会为读者提供关于应该在何处小心使用大数据技术或者结合传统 RDBMS 技术来提供灵活解决方案的指导。

顺着本书的内容结构阅读，我们旨在提供关于学习 MongoDB 和使用 MongoDB 创建应用程序及解决方案的分步指南。

我们衷心希望我们的读者能够享受到阅读本书的乐趣，就像我们享受了编写本书的乐趣一样。

本书内容

- 作为一本指南，将帮助读者领会大数据技术中的各种专业术语并且牢牢掌握大数据的各个方面。
- 作为一本指南，将帮助读者理解 NoSQL 和基于文档的数据库，以及它们与传统的关系型数据库有多么不同。
- 提供了使用 MongoDB 架构解决方案的见解，还提供了 MongoDB 作为一个工具所受限制的信息。
- 系统地介绍了 MongoDB 的架构、开发、管理和数据模型。
- 引用了示例，以便让用户轻松地开始学习该技术。

阅读本书你需要做的准备

MongoDB 支持大多数主流平台。

可以从 MongoDB 下载页面(http://www.mongodb.org/downloads/)上下载 MongoDB 最新稳定的正式版本。

在本书中，我们将专注于在 64 位 Windows 平台上使用 MongoDB，并且在许多地方也引用了如何使用在 Linux 上运行的 MongoDB 的参考。

我们将使用 64 位的 Windows 2008 R2 和 Linux 系统来提供安装过程的示例。

本书读者对象

对于编程人员、大数据架构师、应用程序架构师、技术爱好者、学生、解决方案专家以及那些希望选择合适的大数据产品来满足其需求的人来说，本书将会很有意义。

本书介绍了与大数据、NoSQL 以及在 MongoDB 上架构和开发的详细信息有关的内容。因此它为使用 MongoDB 的开发人员、架构师和运营团队提供了用例。

目　录

第 1 章　大数据 …… 1
1.1　入门指南 …… 1
1.2　大数据 …… 3
1.3　大数据源 …… 4
1.4　大数据的三个 V …… 5
1.4.1　数量 …… 6
1.4.2　多样性 …… 6
1.4.3　速率 …… 7
1.5　大数据的使用 …… 7
1.5.1　可见性 …… 8
1.5.2　发现和分析信息 …… 8
1.5.3　市场细分和产品定制 …… 8
1.5.4　协助决策 …… 8
1.5.5　创新 …… 8
1.6　大数据的挑战 …… 9
1.6.1　政策与程序 …… 9
1.6.2　访问数据 …… 9
1.6.3　技术与技能 …… 9
1.7　传统系统与大数据 …… 10
1.7.1　大数据的结构 …… 10
1.7.2　数据存储 …… 10
1.7.3　数据处理 …… 10
1.8　大数据技术 …… 10
1.9　本章小结 …… 11

第 2 章　NoSQL …… 13
2.1　SQL …… 13
2.2　NoSQL …… 13
2.2.1　定义 …… 14
2.2.2　NoSQL 简史 …… 14
2.3　ACID 对比 BASE …… 15
2.3.1　CAP 定理 …… 15
2.3.2　BASE …… 16
2.4　NoSQL 的优缺点 …… 17
2.4.1　NoSQL 的优点 …… 17
2.4.2　NoSQL 的缺点 …… 18
2.5　SQL 与 NoSQL 数据库的对比 …… 18
2.6　NoSQL 数据库的种类 …… 21
2.7　本章小结 …… 22

第 3 章　MongoDB 介绍 …… 23
3.1　历史 …… 23
3.2　MongoDB 设计原则 …… 24
3.2.1　高速、可扩展性与敏捷性 …… 24
3.2.2　非关系型方法 …… 24
3.2.3　基于 JSON 的文档存储 …… 25
3.2.4　性能与功能对比 …… 25
3.2.5　随处都能运行数据库 …… 25
3.3　与 SQL 的对比 …… 26
3.4　本章小结 …… 26

第 4 章　MongoDB 数据模型 …… 27
4.1　数据模型 …… 27
4.1.1　JSON 和 BSON …… 28
4.1.2　标识符(_id) …… 29
4.1.3　固定集合 …… 30

4.2 多态模式……30
4.2.1 面向对象编程……30
4.2.2 模式演化……31
4.3 本章小结……32

第 5 章 MongoDB-安装与配置……33
5.1 选择你的版本……33
5.2 在 Linux 上安装 MongoDB……33
5.2.1 使用仓储进行安装……34
5.2.2 手动安装……34
5.3 在 Windows 上安装 MongoDB……35
5.4 运行 MongoDB……35
5.4.1 先决条件……35
5.4.2 开启服务……36
5.5 验证安装结果……36
5.6 MongoDB Shell……36
5.7 保障部署安全……37
5.7.1 使用身份验证和授权……37
5.7.2 控制网络访问……42
5.8 使用 MongoDB 云管理器进行配置……46
5.9 本章小结……50

第 6 章 使用 MongoDB Shell……51
6.1 基本查询……51
6.1.1 创建和插入……56
6.1.2 显式创建集合……58
6.1.3 使用循环插入文档……58
6.1.4 通过显式指定_id 进行插入……59
6.1.5 更新……59
6.1.6 删除……61
6.1.7 读取……62
6.1.8 使用索引……68
6.2 进阶介绍……78
6.2.1 使用条件操作符……78
6.2.2 正则表达式……80
6.2.3 MapReduce……81
6.2.4 aggregate()……83
6.3 设计应用程序的数据模型……84
6.3.1 关系型数据模型与标准化……84
6.3.2 MongoDB 文档数据模型方法……86
6.4 本章小结……93

第 7 章 MongoDB 架构……95
7.1 核心程序……95
7.1.1 mongod……95
7.1.2 mongo……95
7.1.3 mongos……96
7.2 MongoDB 工具……96
7.3 独立部署……96
7.4 复制……97
7.4.1 主/从复制……97
7.4.2 副本集……98
7.4.3 实现带有副本集的高级群集……115
7.5 分片……124
7.5.1 分片组件……126
7.5.2 数据分发过程……127
7.5.3 数据平衡过程……130
7.5.4 操作……133
7.5.5 实现分片……134
7.5.6 控制集合分布(基于标签分片)……142
7.5.7 在将数据导入到分片环境时要记住的要点……152
7.5.8 监控分片……153
7.5.9 监控配置服务器……153
7.6 生产环境群集架构……153
7.6.1 场景 1……154
7.6.2 场景 2……155
7.6.3 场景 3……156
7.6.4 场景 4……157
7.7 本章小结……158

第 8 章 MongoDB 阐释……159
8.1 数据存储引擎……159
8.2 (与 MMAPv1 相关的)数据文件……161
8.3 (与 WiredTiger 相关的)数据文件……168
8.4 读取和写入……170
8.5 使用日志时如何写入数据……172
8.6 GridFS——MongoDB 文件系统……176
8.6.1 GridFS 的基本原理……177
8.6.2 GridFS 的底层机制……177
8.6.3 使用 GridFS……179
8.7 索引……182
8.7.1 索引类型……183
8.7.2 行为和限制……188
8.8 本章小结……189

第 9 章 管理 MongoDB……191
9.1 管理工具……191
9.1.1 mongo……191
9.1.2 第三方管理工具……191
9.2 备份和恢复……191
9.2.1 数据文件备份……192
9.2.2 mongodump 和 mongorestore……192
9.2.3 fsync 和锁……196
9.2.4 从备份……198
9.3 导入和导出……198
9.3.1 mongoimport……198
9.3.2 mongoexport……199
9.4 管理服务器……199
9.4.1 启动一台服务器……199
9.4.2 停止服务器运行……200
9.4.3 浏览日志文件……200
9.4.4 服务器状态……201
9.4.5 识别和修复 MongoDB……203
9.4.6 识别和修复集合级别的数据……204
9.5 监控 MongoDB……205
9.5.1 mongostat……205
9.5.2 mongod 网络接口……206
9.5.3 第三方插件……206
9.5.4 MongoDB 云管理器……206
9.6 本章小结……212

第 10 章 MongoDB 用例……213
10.1 用例 1——性能监控……213
10.1.1 模式设计……213
10.1.2 操作……214
10.1.3 分片……218
10.1.4 管理数据……219
10.2 用例 2——社交网络……220
10.2.1 模式设计……220
10.2.2 操作……222
10.2.3 分片……225
10.3 本章小结……225

第 11 章 MongoDB 使用限制……227
11.1 MongoDB 的空间过大(对于 MMAPv1 而言)……227
11.2 内存问题(对于 MMAPv1 而言)……228
11.3 32 位与 64 位对比……228
11.4 BSON 文档……228
11.5 命名空间使用限制……229
11.6 索引使用限制……229
11.7 固定集合使用限制——固定集合中文档的最大数量……229
11.8 分片使用限制……229
11.8.1 及早分片以避免出现问题……230
11.8.2 不能更新分片键……230
11.8.3 分片集合使用限制……230

11.8.4 选择合适的分片键……230
11.9 安全性限制……230
11.9.1 默认情况下没有身份验证……230
11.9.2 与 MongoDB 的交互通信没有被加密……231
11.10 写入和读取限制……231
11.10.1 大小写敏感的查询……231
11.10.2 类型敏感的字段……231
11.10.3 没有联结……231
11.10.4 事务……231
11.11 MongoDB 不适用的范围…232
11.12 本章小结……232
第 12 章 MongoDB 的最佳实践……233
12.1 部署……233
12.1.1 MongoDB 网站的硬件配置建议……234
12.1.2 要注意的一些要点……235
12.2 编码……235
12.3 应用程序响应时间优化……238
12.4 数据安全性……238
12.5 管理……239
12.6 复制延迟……239
12.7 分片……240
12.8 监控……240
12.9 本章小结……241

第 1 章

大 数 据

“大数据是用于描述海量的、具有各种结构并且高速生成的数据的一个术语。这类数据对用于存储和处理数据的传统 RDBMS 系统提出了挑战。大数据为处理和存储数据的新途径铺平了道路。”

在本章中，我们将探讨大数据基础、来源以及挑战，将介绍大数据的三个 V(数量(volume)、速率(velocity)和多样性(variety))以及在面临处理大数据时传统技术所受到的限制。

1.1 入门指南

大数据与云、社交、分析以及移动性一样，都是当前信息技术世界中的流行语。供应给大众使用的互联网和电子设备，其数量每一天都在增长。尤其是，智能手机、社交网络站点以及像平板电脑和传感器这样的其他数据生成设备都在导致数据爆炸性增长。数据是从各种来源中生成的，具有各种格式，例如视频、文本、语音、日志文件以及图片。一个高清(HD)视频的每一秒会生成比单页文本多 2000 倍的字节。

思考一下在 Facebook 公司网站上发表的与该公司有关的以下统计数据：

(1) 2015 年 6 月平均每天有 9 亿 6 千 8 百万日活跃用户。2015 年 6 月平均每天有 8 亿 4 千 4 百万移动端日活跃用户。

(2) 截至 2015 年 6 月 30 日，有 14 亿 9 千万月活跃用户。截至 2015 年 6 月 30 日，有 13 亿 1 千万移动端月活跃用户。

(3) 截至 2013 年 5 月，每天会生成 45 亿个点赞，这一数据相对于 2012 年 8 月增长了 67%。

图 1-1 描述了 Twitter 的统计数据。

这里有另一个示例：想象一下像去看一场电影这样的简单事件所能生成的数据量。你首先要在电影评论网站上搜索一部电影，阅读关于该电影的评论，并且提交查询。你可能会发表关于该电影的推文或者在 Facebook 上发布打算去看该电影的照片。在去电影院的途中，你的 GPS 系统会追踪你的路线并且生成数据。

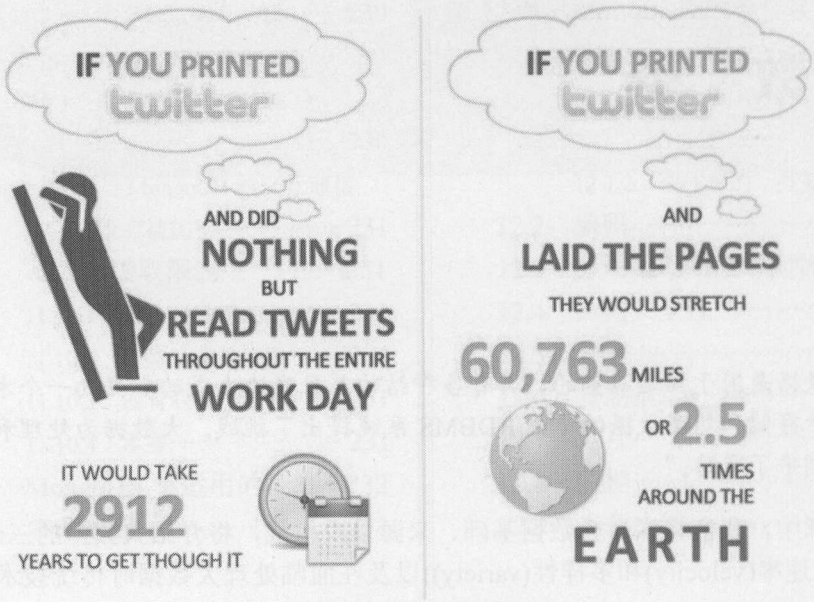

图 1-1 如果将 Twitter 印刷出版……

你会发现这样的场景：智能手机、社交网络站点及其他媒体都在创造数据洪流以便这些公司能够处理和存储。当数据的大小对典型软件工具捕获、处理、存储和管理数据的能力提出挑战时，我们就面临大数据的处理。图 1-2 以图形化的方式定义了大数据。

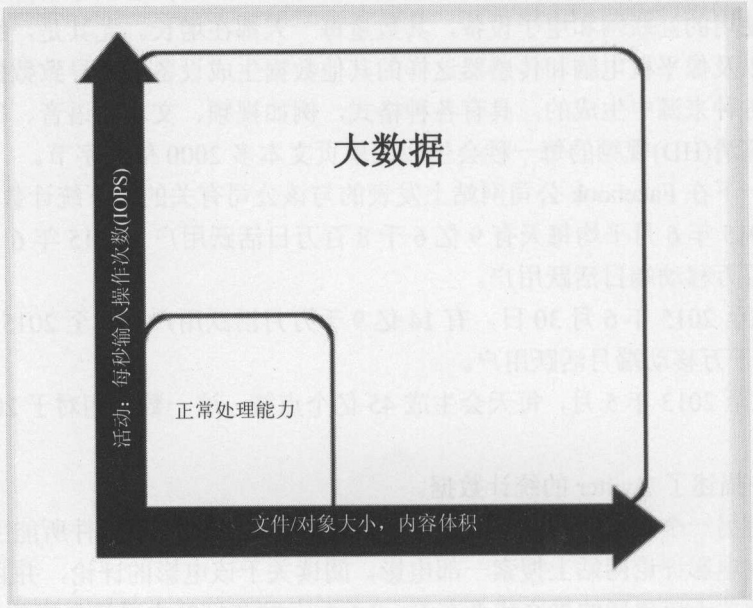

图 1-2 大数据的定义

1.2 大数据

大数据是具有高容量的、高速生成的并且具有多种样式的数据。我们来看看大数据的一些实际情况。

大数据的一些实际情况

世界各地的各种研究团队已经对所产生的大量数据进行了分析。例如，IDC 的分析表明，一年中(2007 年)所产生的数字数据量要比整个世界用于存储它的总体容量还大，这意味着没有办法存储所产生的所有数据。另外，数据产生的速度很快就会超过数据存储能力扩充的速度。

后面几节涵盖了来自 MGI(Mckinsey Global Institute，麦肯锡全球研究所)于 2011 年 5 月发布的报告(www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation)中的见解。该研究明确地表明，大数据的商业和经济潜力及其广阔的影响是很重要的问题，商业领袖和政策制定者必须应对该问题。

1. 大数据的大小因行业而异

大数据的增长是在每一个行业中都被观测到的一种现象。MGI 估算出，2010 年世界各地的企业使用了超过 7 艾字节(exabyte)的增额硬盘驱动数据存储能力；有意思的是，其中近 80%的数据看起来都是在其他地方被存储过的重复数据。MGI 还估算出，截至 2009 年，美国经济中涉及的所有行业几乎每家公司平均都有至少 200 太字节(terabyte)的存储数据，并且许多行业中每家公司平均都有超过 1 拍字节(petabyte)的存储数据。

有些行业展现出了远高于其他行业的数据强度水平；在这里，数据强度指的是该行业中跨公司/企业所累积的数据的平均量，这表明这些公司/企业拥有更多的从大数据中获得价值的潜力。

金融服务行业，其中包括银行、投资以及安全服务，都是高度交易导向的；根据法规，它们也被要求存储数据。该分析表明，从平均值来看它们中的每家企业存储了最大的数字数据量。

通信和媒体公司、公用事业以及政府，这些领域中的每家企业或组织同样存储了大量的数字数据，这似乎反映出，像这样的实体具有大量的操作和多媒体数据。

离散型和流程式制造业具有以字节方式存储的最高水平的聚合数据。不过，这些行业在强度方面的排名要远低得多，因为它们都被划分成了大量的企业。

2. 大数据的类型因行业而异

该 MGI 研究还表明，数据存储的类型也会因行业而异。例如，零售业和批发业、政府的行政管理部门以及金融服务都会产生大量的文本和数值数据，其中包括客户数据、交易信息以及数学建模和模拟。像制造、医疗卫生、媒体和通信这样的行业都要负责处理和存储较高比例的多媒体数据。而形如 X 光、CT 和其他扫描的图片数据在医疗卫生

行业中占用了其数据存储容量。

在大数据的地理性传播方面，目前北美和欧洲占据了全球总量的 70%。幸亏有云计算，一个地区产生的数据才能被存储到另一个国家的数据中心。因此，拥有大量云和主机服务提供商产品的国家往往具有大量的数据存储。

1.3　大数据源

在本节中，我们将介绍造成数据大小日益增长的主要因素。图 1-3 描述了产生数据的主要来源。

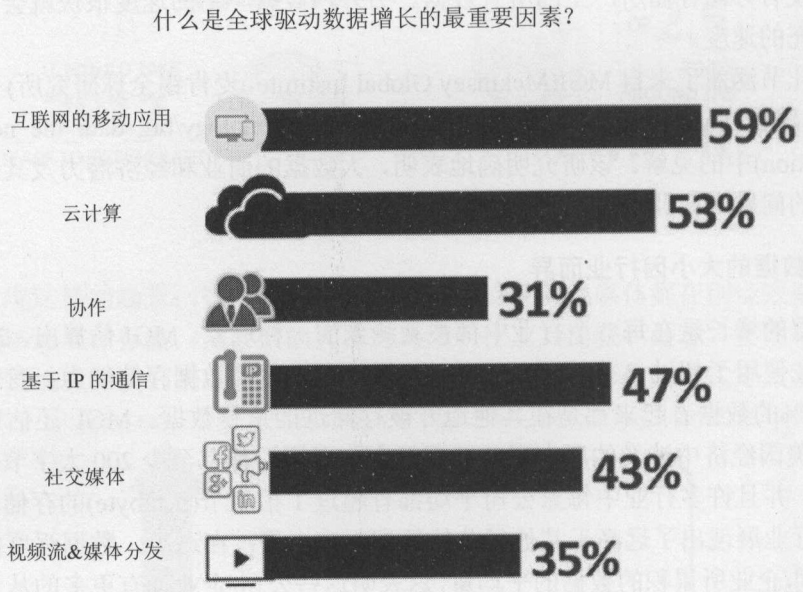

图 1-3　数据源

正如 MGI 报告中所强调的，这些数据的主要来源是：

- 企业，如今它们正在收集更多粒度的数据，其中包含每笔交易的更多详细信息，以便理解消费者的行为。
- 跨行业多媒体使用的增长，例如医疗卫生、商品量产公司等。
- 日益流行的社交媒体站点，例如 Facebook、Twitter 等。
- 智能手机的迅速普及，它们使得用户积极地使用社交媒体站点以及其他互联网应用。
- 日常生活中传感器和设备使用的增加，网络将它们与计算资源连接在一起。

MGI 报告还凸显出，像传感器这样的机器对机器设备(这也被称作物联网或 IoT(Internet of Things))的数量在下一个 5 年中每年将增长 30%。

因此，数据的增长速度正在提高，并且其多样性也是如此。另外，数据产生的模型已经从一种模式(一些公司产生数据，而其他公司消费这些数据)转变为另一种模式(每个

人都产生数据，而每个人也都消费这些数据)。这是由于消费者信息技术和互联网技术伴随着像社交媒体这样的潮流而渗透到人们的生活中所造成的。图 1-4 描述了数据产生模型中的变化。

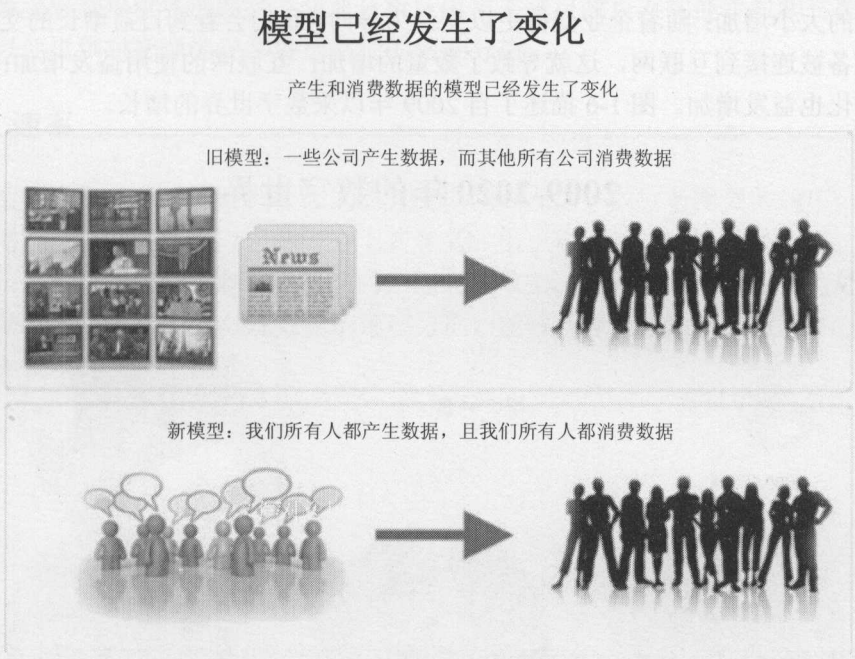

图 1-4 数据模型

1.4 大数据的三个 V

我们已经用三个 V 定义了大数据：数量、速率以及多样性，如图 1-5 所示。我们来看看这三个 V。组织和 IT 领袖专注于这些方面是势在必行的。

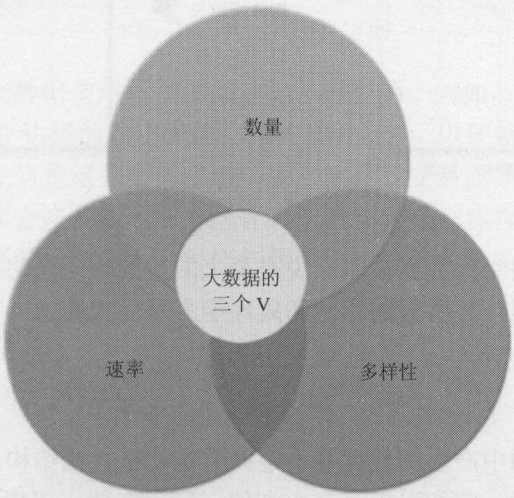

图 1-5 大数据的三个 V，“大”并不仅仅指数量

1.4.1　数量

大数据中的数量意味着数据的大小。正如前面几节中所探讨过的，各种因素都会造成大数据的大小增加：随着企业变得更以交易为导向，我们会看到日益增长的交易数量；更多的设备被连接到互联网，这就导致了数量的增加；互联网的使用益发增加；并且内容的数字化也益发增加。图 1-6 描述了自 2009 年以来数字世界的增长。

图 1-6　数字世界的大小

在如今的情形下，数据并非仅仅产生自企业内部；它还基于与更多的企业和客户进行的交易而产生。这就要求企业维护大量的客户数据。现今拍字节的规模正变得越来越常见。图 1-7 描述了数据增长速率。

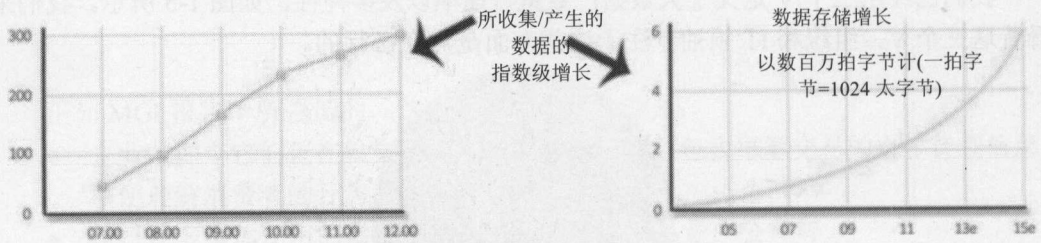

图 1-7　增长速率

这样大量的数据对于大数据技术来说是最大的挑战。用及时有效且低成本的方式来存储、处理和访问数据，其所需的存储和处理能力是巨大的。

1.4.2　多样性

从各种设备和来源中产生的数据并不遵循固定的格式或结构。相较于文本，CSV 或 RDBMS 数据多种多样，它们可以是文本文件、日志文件、视频流、照片、仪表读数、

股票报价器数据、PDF、音频以及各种其他非结构化格式。

如今无法对数据的结构进行控制。新的数据来源和结构正在被飞速创建。因此技术上的责任在于找到一个解决方案来分析和可视化所存在的大量各式各样的数据。举例来说，要为上班族提供备选路径，交通分析应用需要来自数百万智能手机和传感器的数据反馈，以便提供对交通情况和备选路径的精准分析。

1.4.3 速率

大数据中的速率是指数据被创造的速度以及处理这些数据所需要的速度。如果无法按照所需的速度处理数据，它就失去了其意义。由于数据是在社交媒体站点、传感器、股票报价器、计量器和检测仪中流动的，因此当数据流动和静止时组织快速处理数据就非常重要(参见图 1-8)。在处理数据的速率方面，能够足够快速地做出反应并且进行处理是大数据技术的另一个挑战。

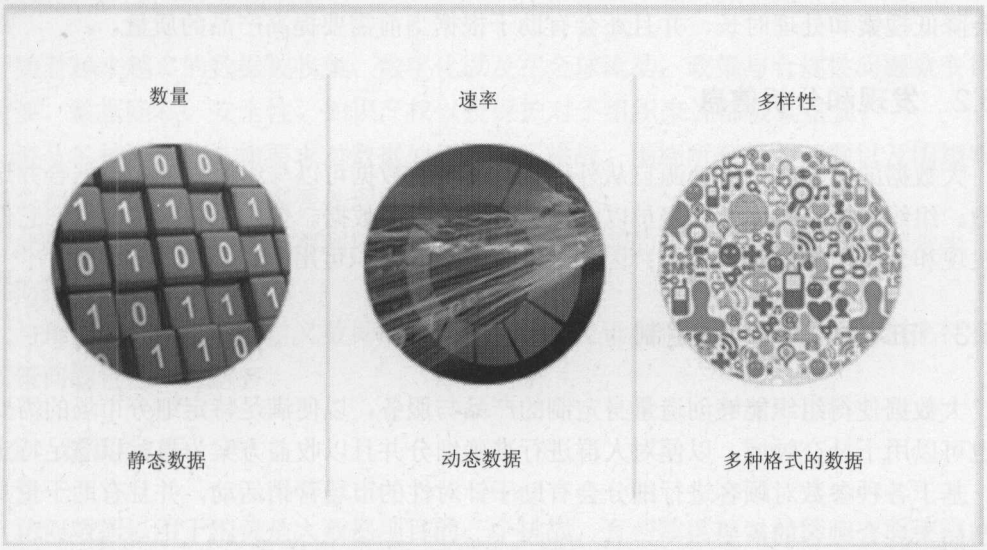

图 1-8 数据的三个方面

实时领悟在许多大数据使用场景中都是必不可少的。例如，一个算法交易系统会使用来自市场和像 Twitter 这样的社交媒体站点的实时反馈，以便就股票交易做出决策。对这些数据进行处理的任何延迟都会意味着错失一笔股票交易而导致的数百万美元损失。

在讨论大数据时都会谈及第 4 个 V。这第 4 个 V 就是真实性(veracity)，它意味着并非所有的数据都是重要的，因此识别出哪些数据能够提供有意义的见解，哪些应该被忽略是至关重要的。

1.5 大数据的使用

本节将专注于将大数据用于为组织创造价值的各种方式。在我们深入探究如何才能

让大数据变得对组织可用之前，我们首先看看为何大数据很重要。

大数据是全新的数据来源；它是你发表一篇博文时所产生的数据，比如发表关于一个产品或旅行的博文。以前，如此细枝末节般的可用信息并不会被捕获。现在这些信息会被捕获，并且拥抱这些数据的组织可以寻求创新、提高其敏捷性，以及提升其盈利能力。

大数据可以各种方式为任何组织创造价值。正如 MGI 报告中所列示的，这一点可以被广泛归纳为大数据的 5 种使用方式。

1.5.1 可见性

让利益相关方能够及时地访问数据会产生大量的价值。我们用一个示例来理解这一点。思考一家制造业公司，其研发部、工程部以及生产制造部分布在不同地理位置的情况。如果可跨所有这些部门对其数据进行访问，并且其数据可以被快速整合，那么就不仅会降低搜索和处理时长，并且还会有助于根据当前需要提高产品的质量。

1.5.2 发现和分析信息

大数据的大部分价值都源自从外部来源收集的数据可以与组织内部的数据合并这一处理。组织在捕获与库存、雇员以及顾客有关的详尽数据。使用所有这些数据，它们就能发现和分析新的信息与模式；因此，这一信息和知识可用于提升处理和性能。

1.5.3 市场细分和产品定制

大数据使得组织能够创造量身定制的产品与服务，以便满足特定细分市场的需要。这也可以用于社交领域，以便对人群进行准确细分并且以收益方案为目标以满足特定需要。基于各种参数对顾客进行细分会有助于针对性的市场营销活动，并且有助于量身定制产品来迎合顾客的需要。

1.5.4 协助决策

大数据可以大幅降低风险、改进决策、并且揭示有价值的见解。信用卡处理中的自动化欺诈警报系统以及库存的自动调整都是基于大数据分析提供协助或自动化决策的系统示例。

1.5.5 创新

大数据会以产品和服务的形式促成新理念的创新。它会促成现有产品或服务中的创新以便触及大部分人。使用为实际产品收集到的数据，生产商不仅可以进行创新以便创造下一代产品，并且它们还能够对产品销售进行创新。

举个例子，可以分析来自机器和汽车的实时数据来为维护计划提供见解；可以监控

机器的磨损来制造更具复原能力的机器；油耗监控可以产生更高效的引擎。实时交通信息已经让上班族的日常生活更加容易，这是通过为他们提供备选路径的选项来实现的。

因此，大数据并不仅仅指数据的数量。它带来了从日益增长的数据池中找出有意义的见解的机会。它正在帮助组织做出更为明智的决策，这能让它们更加灵活。它不仅为组织提供了做出明智决策来强化现有业务的机会，并且还有助于发现新的商机。

1.6 大数据的挑战

大数据也带来了一些挑战。在本节中，我们将重点介绍其中几个挑战。

1.6.1 政策与程序

随着越来越多的数据被收集、数字化以及在全球流动，政策与合规性问题就变得愈加重要。数据隐私、安全性、知识产权以及保护对于组织来说都极其重要。

遵从各种法定和法律要求对数据处理提出了挑战。围绕所有权的问题以及围绕数据的债务都是大数据情况下需要处理的重要法律问题。

此外，许多大数据项目都利用了公共云计算提供商的可伸缩性功能，这就带来了合规性的挑战。

与谁拥有数据、如何定义数据的合理使用，以及谁负责数据的准确性和机密性有关的政策问题也需要被解答。

1.6.2 访问数据

访问数据以用于消费是大数据项目的一个挑战。有些数据可以被第三方所使用，而访问权会是一个法律上的、合同性的挑战。

可以在 Facebook、Twitter 源、评论以及博客上访问到关于一个产品或一项服务的数据，那么该产品的所有者如何从各个提供商所拥有的各种源中访问这些数据呢？

同样，合同条款和访问大数据的经济诱因需要被捆绑起来以便让数据可以被消费者所使用。

1.6.3 技术与技能

必须利用专门为应对大数据需要而构建的新工具和技术，而非尝试通过传统系统来处理前面提及的问题。在处理大数据方面，传统系统的缺陷是一方面，而对于较新的技术缺乏经验丰富的资源则是一个挑战，这是任何大数据项目都必须处理的问题。

1.7 传统系统与大数据

在本节中，我们将探讨组织在面临使用传统系统管理大数据时的挑战。

1.7.1 大数据的结构

传统系统旨在处理结构化数据，其中带有列的表都是定义好的。保存在列中的数据格式也是预先就知道的。

不过，大数据是具有许多结构的数据。它基本上是非结构化数据，比如图片、视频、日志等。

由于大数据可以是非结构化的，因此被创建以执行快速查询和分析的传统系统就无法被用于保存或处理大数据，因为这些系统都是通过像基于保存在各个列中的特定数据类型进行索引的技术来执行的。

1.7.2 数据存储

传统系统使用了大型服务器以及 NAS 和 SAN 系统来存储数据。随着数据的增长，就必须增加服务器的大小以及后端存储的大小。传统的旧式系统通常运行于可纵向扩展的模型中，需要为一台服务器添加越来越多的计算资源、内存和存储来满足日益增长的数据需求。因此处理时长会呈指数级增长，这就无法满足大数据的另一个重要需求，也就是速率。

1.7.3 数据处理

传统系统中的算法旨在处理结构化数据，比如字符串和整数。它们也受到数据大小的限制。因此，传统系统无法应对非结构化数据和大量此类数据的处理，以及需要执行的处理所需达到的速度。

因此，要从大数据中获得价值，我们需要在存储、计算和检索方面利用较新的技术，并且我们需要将新的技术用于数据分析。

1.8 大数据技术

我们已经向你介绍了什么是大数据。在本节中，我们将简要查看哪些技术可以处理这样庞大的数据源。要探讨的技术需要有效接受和处理不同类型的数据。

以下是让组织能够最大限度地利用其大数据的最新技术进步：

(1) 特别为大型非结构化数据设计的新的存储和处理技术

(2) 并行处理

(3) 群集

(4) 大型网格环境
(5) 高连通性和高吞吐量
(6) 云计算和横向扩展架构

越来越多的技术都在利用这些技术进步。在本书中，我们将探讨 MongoDB，可用于存储和处理大数据的其中一种技术。

1.9 本章小结

在本章中，你学习了与大数据有关的知识。你了解了产生大数据的各种源，以及大数据所带来的应用和挑战。你还了解了为何需要较新的技术来存储和处理大数据。

在后面的章节中，将了解一些有助于组织管理大数据并且让组织能够从大数据中获得有意义的见解的技术。

第 2 章

NoSQL

“NoSQL 是设计互联网规模数据库解决方案的一种新方式。它并非一个产品或一项技术，而是定义一套数据库技术的术语，它并不以传统的 RDBMS 原则为基础。”

在本章中，我们将讲解 NoSQL 的定义和基础知识。我们将介绍 CAP 定理并且探讨 NRW 表示法。我们将对比 ACID 和 BASE 方法，最后以 NoSQL 和 SQL 数据库技术的对比来结束本章。

2.1 SQL

RDBMS 的概念源自 E.F.Codd 发表于 1970 年的标题为“用于大型共享数据银行的数据关系模型”的白皮书。用于查询 RDBMS 系统的语言就是 SQL(结构化查询语言，Sequel Query Language)。

RDBMS 系统非常适合于保存在列和行中的结构化数据，可以使用 SQL 查询这些数据。RDBMS 系统基于 ACID 事务的概念。ACID 代表原子性(Atomic)、一致性(Consistent)、隔离性(Isolated)、持久性(Durable)，其中

- 原子性意味着要么一个事务的所有变更都完全被应用，要么全都不被应用。
- 一致性意味着在应用事务之后数据处于一致性状态。这意味着在一个事务被提交之后，提取特定数据的查询将得到相同的结果。
- 隔离性意味着被应用到相同数据集的事务都是彼此独立的。因此，一个事务将不会干扰另一个事务。
- 持久性意味着变更在系统中是永久性的，并且即使出现任何故障也不会丢失。

2.2 NoSQL

NoSQL 是用于指代非关系型数据库的一个术语。因此，它包含了大多数不以常规的 RDBMS 原则为基础的数据存储，并且被用于处理互联网规模的大数据集。

正如上一章中所探讨过的，大数据对像 RDBMS 系统这样的存储和处理数据的传统方式提出了挑战。因此，我们看到了 NoSQL 数据库的崛起，它旨在在有限的时间和成本内处理这样的大量且多样的数据。

因此，NoSQL 数据库是从处理大数据的需要中发展而来的；传统的 RDBMS 技术无法提供合适的解决方案。图 2-1 显示了相较于结构化数据，这些年来非结构化数据/半结构化数据数量的上升。

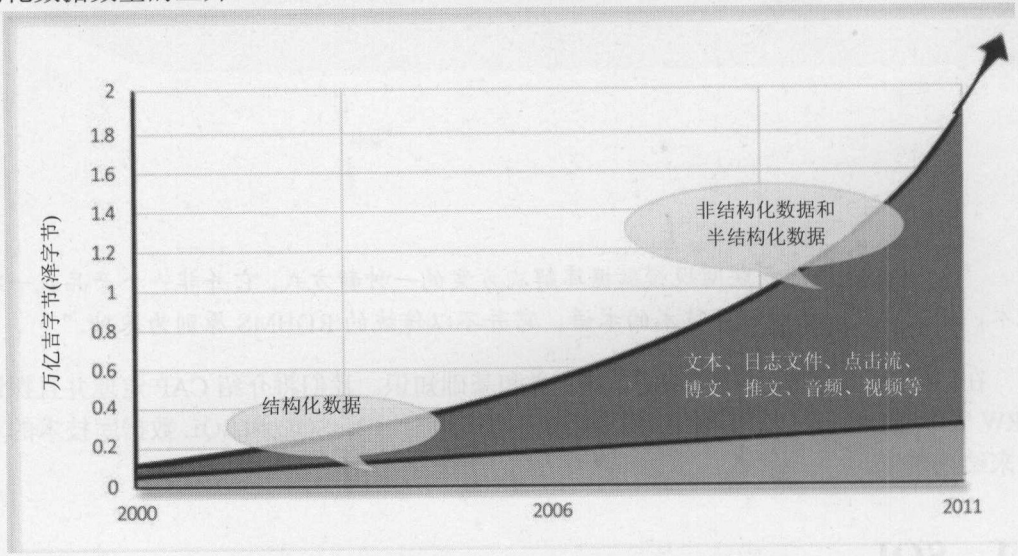

图 2-1 结构化数据对比非结构化数据/半结构化数据

以下是适合 NoSQL 数据库的大数据使用场景的一些示例：

- **社交网络图：**谁与谁有关联？谁的文章应该在一个社交网络站点的用户涂鸦墙或主页上可见？
- **搜索和检索：**搜索具有特定关键字的所有相关页面，按照页面上该关键字出现的次数排序。

2.2.1 定义

NoSQL 并没有一个正式的定义。它代表从根本上与 RDBMS 不同的一种持久化/数据存储机制的形式。不过如果非要定义 NoSQL 的话，应该是这样的：NoSQL 是用于数据存储的一个涵盖性术语，它不遵循 RDBMS 原则。

提示：

最初该术语被用于表示“如果想要扩展，则不要使用 SQL”。之后这个术语被重新定义为“不仅仅是 SQL”，这意味着除了 SQL，还存在其他备受赞誉的数据库解决方案。

2.2.2 NoSQL 的简史

在 1998 年，Carlo Strozzi 首创了 NoSQL 这个术语。他使用此术语区分他的数据库，因为该数据库没有 SQL 接口。2009 年初该术语再次出现，当时 Eric Evans(Rackspace 的一位雇员)在一次关于开源分布式数据库的活动中使用这个术语来提及非关系型且不遵

循关系型数据库的 ACID 特性的分布式数据库。

2.3 ACID 对比 BASE

在前面的介绍中我们提到过，传统的 RDBMS 应用程序专注于 ACID 事务。无论这些特性看起来有多么必要，它们都不符合 Web 规模应用程序的可用性和性能需求。

例如，我们假设你拥有一家像 OLX 这样的公司，该公司销售像未使用过的家用物品(旧家具、汽车等)这样的商品，并且使用 RDBMS 作为其数据库。我们思考两个场景。

第一个场景：我们看看一个电子商务购物网站，其中一个用户正在购买一个商品。在该笔交易期间，该用户会锁定数据库的一部分，即库存，而其余的每一个用户都必须等待造成锁定的该用户完成其交易。

第二个场景：应用程序可能最终会使用缓存数据或者未锁定的记录，从而导致不一致性。在这种情况下，两个用户可能最终都会购买该商品，而此时其库存实际已经变成零了。

该系统可能会变得很慢，从而影响可扩展性和用户体验。

与传统 RDBMS 系统的 ACID 方法相反，NoSQL 解决了这个问题，它使用了一种通常被称为 BASE 的方法。在阐释 BASE 之前，我们先来探究 CAP 定理的概念。

2.3.1 CAP 定理

Eric Brewer 于 2000 年提出了 CAP 定理(布鲁尔定理，Brewer's Theorem)。这是一个重要的概念，处理分布式数据库的开发人员和架构师需要很好地理解它。该定理规定，在设计一个分布式环境中的应用程序时，存在三种基本需求，分别是一致性、可用性以及分区容错性。

- **一致性**意味着在修改数据的任何操作被执行之后，数据仍旧保持一致，并且所有访问该应用程序的用户或客户端都要得到相同的更新后的数据。
- **可用性**意味着系统总是保持可用。
- **分区容错性**意味着，即使系统被划分成无法彼此通信的几组服务器，也要持续地正常运行。

CAP 定理宣称，在任何时间点，一个分布式系统都只能满足上面三个保障中的两个，如图 2-2 所示。

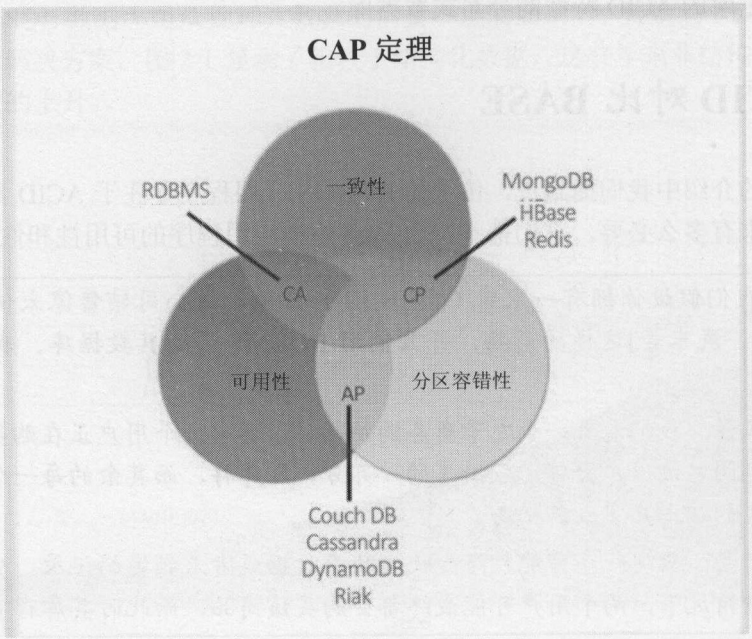

图 2-2 CAP 定理

2.3.2 BASE

Eric Brewer 提出了 BASE 这个缩略语。BASE 可被解释为：

- **基本可用**(Basically Available)意味着根据 CAP 定理，系统将是可用的。
- **软状态**(Soft state)表明，即便没有为系统提供任何输入，其状态也将随时间而变化。这是符合最终一致性的。
- **最终一致性**(Eventual consistency)意味着从长远来看，系统将达到一致性，即便这段时间内没有任何输入被发送到系统也会如此。

因此，BASE 与 RDBMS ACID 事务大不相同。

我们已经介绍过 NoSQL 数据库是最终一致的，但最终一致的实现可能会因不同的 NoSQL 数据库而不同。

NRW 是用于描述最终一致性模型是如何在所有 NoSQL 数据库中实现的表示法，其中：

- **N** 是数据库已经保留的数据副本数量。
- **R** 是在返回一个读取请求的输出之前，一个应用程序需要涉及的副本数量。
- **W** 是在一个写操作被标记为成功完成之前需要对其进行写入的数据副本数量。

使用这些表示法配置，数据库就实现了最终一致性的模型。

可以同时在读和写操作层面实现一致性。

写操作

N=W 表明，在将控制返回给客户端并且将写操作标记为成功之前，写操作将更新所有的数据副本。这类似于实现同步复制时传统 RDBMS 数据库的工作方式。这个设置会

降低写性能。

如果写性能是一个关注点，则意味着你希望写操作能快速完成，那么可以设置 W=1、R=N。这表明，写操作将仅更新任意一个副本，并且将写操作标记为成功，但无论何时用户发出一个读请求，它都会读取所有的副本以返回结果。如果这些副本的任何一个没有被更新过，那么它将确保做同样的更新，只有这样读操作才会成功。这一实现会降低读性能。

因此，大多数 NoSQL 实现都使用 N>W>1。这表明需要成功更新多个节点；不过，并非所有节点都需要同时更新。

读操作

如果 R 被设置为 1，那么读操作将读取任意一个数据副本，它可能是过期的。如果 R>1，那么将读取多个副本，并且会读取最近的值。不过，这会降低读操作的性能。

使用 N<W+R 总是会确保一个读操作检索最新的值。这是由于写副本和读副本的数量总是大于实际的副本数量，从而确保至少有一个读副本具有最新的版本。这是约定俗成的组合。

表 2-1 对比了 ACID 和 BASE。

表 2-1 ACID 对比 BASE

ACID	BASE
原子性	基本可用
一致性	最终一致性
隔离性	软状态
持久性	

2.4 NoSQL 的优缺点

在本节中将介绍 NoSQL 数据库的优缺点。

2.4.1 NoSQL 的优点

我们来谈谈 NoSQL 数据库的优点。

- **高扩展性：**此纵向扩展方法在事务率和快速响应需求增加时会失败。与此相反，新一代的 NoSQL 数据库旨在横向扩展(例如，使用低端廉价服务器进行水平扩展)。
- **可维护性和管理运营：**NoSQL 数据库主要旨在处理自动修复、分布式数据以及较简单的数据模型，这会导致低水平的可维护性和管理运营。

- **低成本**：NoSQL 数据库的目的通常在于使用一个廉价服务器的群集，以便让用户可以花费较低的成本来存储和处理更多的数据。
- **灵活的数据模型**：NoSQL 数据库拥有一个非常灵活的数据模型，这使得它们可以处理任何类型的数据；它们并不遵从死板的 RDBMS 数据模型。因此，任何涉及更新数据库模式的应用程序变更都能被轻易实现。

2.4.2 NoSQL 的缺点

除了上述优点之外，在开始使用这些平台开发应用程序之前，你也需要知道存在许多障碍。

- **成熟度**：大多数 NoSQL 数据库都是试生产版本，其关键功能仍然有待实现。因此，在选择一个 NoSQL 数据库时，你应该彻底分析该产品以确保其功能已完全实现而不是仍然停留在待处理清单上。
- **支持度**：支持度是需要考虑的一个限制条件。大多数 NoSQL 数据库都来自曾经开源的初创公司。因此，相较于企业级软件公司，其支持度极小，并且可能没有全球影响力或支持资源。
- **有限的查询功能**：由于 NoSQL 数据库通常是被开发出来以满足网络规模应用程序的扩展需求的，它们提供了有限的查询功能。一个简单查询需求可能涉及很多编程专业知识。
- **管理运营**：尽管 NoSQL 旨在提供无管理解决方案，但它仍然要求安装和维护该解决方案的技能和精力。
- **专业知识**：由于 NoSQL 是一个发展中的领域，因此与其技术有关的专业知识局限于开发人员和管理员社区。

尽管 NoSQL 正在变成数据库领域的一个重要部分，但需要意识到其产品的局限性和优势，以便能够正确选择 NoSQL 数据库平台。

2.5 SQL 与 NoSQL 数据库的对比

现在你知道了关于 NoSQL 数据库的详细信息。尽管 NoSQL 正越来越多地被用作一种数据库解决方案，但它并不是为了取代 SQL 或 RDBMS 数据库。在本节中，将看到 SQL 和 NoSQL 数据库之间的区别。

我们来快速回顾一下 RDBMS 系统。RDBMS 系统已经流行了近 30 年，即便现在，它们对于应用程序数据存储的解决方案架构师来说仍然是首选。如果我们要列出 RDBMS 系统的一些优点，那么最主要的就是 SQL 的使用，它是用于数据处理的富声明式查询语言。它能被用户很好地理解。此外，RDBMS 系统为事务提供了 ACID 支持，这在许多行业中都是必要的，比如银行业应用程序。

不过，RDBMS 系统的最大缺点就是随着数据的增长，其难以处理模式变更和扩展问题。随着数据的增长，其读/写性能会降低。你在使用 RDBMS 系统时会面临扩展问题，

这是因为它们主要旨在纵向扩展而非横向扩展。

与 SQL RDBMS 数据库相反，NoSQL 促进了数据存储的发展，它摆脱了 RDBMS 的范式。

我们来探讨一些技术场景以及它们在 RDBMS 和 NoSQL 中的对比情况：

- **模式灵活性：**这对于将来轻易地进行功能增强和集成外部应用程序(出站或入站)来说是必要的。

  RDBMS 在其设计上是非常不灵活的。添加一个列是绝对的禁忌，尤其是在表已经有一些数据的时候。这种情况可能是由于默认值、索引或性能影响造成的。通常，你最终会创建新的表，从而由于引入跨这些表的关系而导致复杂性增加。
- **复杂查询：**表的传统设计会导致开发人员编写复杂的 JOIN 查询，这不仅难以实现和维护，而且也会消耗大量的数据库资源来执行。
- **数据更新：**跨表更新数据大概是其中一个较为复杂的场景，当它们是事务的一部分时尤为如此。要注意，长时间保持事务的开启状态会损害到性能。你还必须为将更新传播到系统的多个节点进行规划。而如果系统不支持多个主服务器或者同时写多个节点的话，那么就存在节点故障以及整个应用程序迁移为只读模式的风险。
- **可扩展性：**通常可能需要的唯一可扩展性就是为了读操作。不过，随着操作的增长，有几个因素会影响这个速度。有一些关键的问题需要回答：
  - 跨物理数据库实例同步数据需要多少时间？
  - 跨数据中心同步数据需要多少时间？
  - 同步数据需要多大的带宽？
  - 交换的数据是否已经优化过？
  - 跨服务器同步任意更新时的延迟是多少？通常，一次更新期间，记录将被锁定。

  基于 NoSQL 的解决方案为上面列出的大多数挑战提供了答案。

  我们现在来看看 NoSQL 必须为上面提到的每个技术问题提供什么。
- **模式灵活性：**面向列的数据库会将数据存储为列，这与 RDBMS 中的行相反。这使得所要求的在运行时添加一列或多个列的灵活性成为可能。类似的，允许存储半结构化数据的文档存储也是好的选择。
- **复杂查询：**NoSQL 数据库不支持关系或外键。这里不能存在复杂查询，也没有 JOIN 语句。

  这是一个缺点吗？一个查询如何跨表？

  这肯定是一个功能性的缺点。要跨表查询，就必须执行多个查询。一个数据库就是一个共享资源，它被跨应用程序服务器使用并且必定不能尽可能快地从使用中释放出来。可供选择的方案涉及简化要执行的查询、缓存数据以及在应用层中执行复杂操作这一组合。很多数据库都提供了内置的实体级别缓存。这意味着在访问一条记录时，数据库可能会透明地自动缓存它。该缓存可以是用于性能和扩展的内存式分布缓存。

- **数据更新：**数据更新和跨物理实例的同步是要解决的难点工程问题。相较于跨多个数据中心进行同步，在一个数据中心内跨节点同步面临一组不同的要求。可能是希望最好将延迟限制在几毫秒或几十毫秒内。NoSQL 解决方案提供了非常棒的同步选项。

  例如，MongoDB 允许跨节点并发更新、具有冲突解决方案的同步，以及最终实现在可接受的时长内运行几毫秒并确保跨数据中心的一致性。因此，MongoDB 没有隔离的概念。注意，现在因为管理事务的复杂性可能被移出了数据库，所以应用程序就必须承担一些困难的工作。

  例如，在实现事务时的两阶段提交(http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits/)。

  许多数据库都提供多版本并发控制(Multiversion Concurrency Control，MCC)以达到事务一致性。

  按照 eBay 技术研究员 Dan Pritchett(www.addsimplicity.com/)的说法，eBay.com 没有使用事务。注意 PayPal 也没有使用事务。
- **可扩展性：**出于明显的原因，NoSQL 解决方案提供了更大的可扩展性。面向事务的 RDBMS 所要求的大量复杂性在不遵从 ACID 的 NoSQL 数据库中并不存在。有意思的是，因为 NoSQL 不提供跨表引用以及不可能使用 JOIN 查询，并且由于你无法编写一条查询来跨多个表整理数据，所以一个简单而合理的解决方案就是——也常常会这样做——跨表复制数据。在一些场景中，将信息嵌入主实体中——尤其是在一对一映射的情况下——可能会是一个好办法。

表 2-2 比较了 SQL 和 NoSQL 技术。

表 2-2　SQL 与 NoSQL 对比

	SQL 数据库	NoSQL 数据库
类型	所有类型都支持 SQL 标准	存在多种类型，比如文档存储、键值存储、列数据库等
发展历史	开发于 1970 年	开发于 2000 年
示例	SQL Server、Oracle、MySQL	MongoDB、HBase、Cassandra
数据存储模型	数据被存储在表的行和列中，其中每一列都有一个特定类型 表通常都是按照标准化原则创建的 使用联接来从多个表中检索数据	数据模型取决于数据库类型。比如数据被存储为键值对以用于键值存储。在基于文档的数据库中，数据会被存储为文档 数据模型是灵活的，与 RDBMS 死板的表模型相反
模式	固定的结构和模式，因此对模式的任何变更都涉及修改数据库	动态模式，通过扩展或修改当前模式就能适应新的数据类型或结构 可以动态添加新的字段

(续表)

	SQL 数据库	NoSQL 数据库
可扩展性	使用了纵向扩展方式；这意味着随着负荷的增加，需要购买更大、更贵的服务器来容纳数据	使用了横向扩展方式；这意味着将数据负荷分散到廉价服务器上
支持事务	支持 ACID 和事务	支持分区和可用性，会损害事务 事务存在于某个级别，比如数据库级别或文档级别
一致性	强一致性	取决于产品。有些产品选择提供强一致性，而有些提供最终一致性
支持度	提供了高力度的企业级支持	开源模型。通过构建该开源产品的第三方或公司提供支持
成熟度	已经存在很长时间了	其中有些是成熟的；其他的还处于发展阶段
查询功能	可通过易用的 GUI 界面来使用	查询可能需要编程专业技术和知识。与 UI 不同，其重心在于功能和编程接口
专业知识	那些已经利用 SQL 语言和 RDBMS 概念来架构和开发应用程序的开发人员的大型社区	致力于这些开源工具的开发人员的小社区

2.6 NoSQL 数据库的种类

在本节中将快速探究 NoSQL 领域，将查看 NoSQL 数据库不断出现的类别。表 2-3 显示了 NoSQL 领域中的一些项目，其中有每个类别的类型和产品。

表 2-3 NoSQL 类别

类 别	简 要 描 述	产 品 示 例
基于文档的	以文档形式存储数据 例如，{Name="Test User", Address="Address1", Age:8}	MongoDB
XML 数据库	将 XML 用于存储数据	MarkLogic
图形数据库	数据被存储为节点集合。这些节点通过边界被连接起来。一个节点相当于编程语言中的一个对象	GraphDB
键值存储	将数据存储为键值对	Cassandra、Redis、memcached

NoSQL 数据库是以数据存储的方式为基础来分类的。NoSQL 主要遵循一种水平结构，因为通常需要近乎实时地从大量的数据中提供决策信息。它们的优化是为了使用内置的功能大规模进行插入和检索操作以便复制和群集。

表 2-4 简要提供了 NoSQL 数据库各种类别之间的功能比较。

表 2-4 功能对比

功能	面向列	文档存储	键值存储	图形
类似表模式的支持(列)	是	否	否	是
完全更新/抓取	是	是	是	是
部分更新/抓取	是	是	是	否
对值进行查询/过滤	是	是	否	是
跨行聚合	是	否	否	否
实体之间的关系	否	否	否	是
跨实体视图支持	否	是	否	否
批量抓取	是	是	是	是
批量更新	是	是	是	否

在考虑使用一个 NoSQL 项目时，一件重要的事情是与你相关的功能集。在决定选用一个 NoSQL 产品时，你首先需要非常谨慎地理解问题需求，然后你应该看看其他已经使用过该 NoSQL 产品解决类似问题的人。要记住，NoSQL 仍然处于发展期，因此需要向其他使用 NoSQL 的人学习，从而做出更好的选择。

此外，还需要考虑以下问题。

- 需要处理的数据有多大？
- 读和写可接受的吞吐量是多少？
- 系统中实现了多大程度的一致性？
- 系统需不需要支持高写入性能或高读取性能？
- 维护和管理运营的容易程度如何？
- 需要查询什么？
- 使用 NoSQL 的好处是什么？

我们建议你从小处但重要的部分入手，并且在可能的地方考虑使用混合方式。

2.7 本章小结

在本章中，你学习了与 NoSQL 有关的知识。现在你应该理解什么是 NoSQL 以及它与 SQL 有多大区别。你还了解了 NoSQL 的各种类别。

在后面几章中将介绍 MongoDB，它是一个基于文档的 NoSQL 数据库。

第 3 章

MongoDB 介绍

“MongoDB 是一款领先的 NoSQL 文档存储数据库。它使得组织可以处理大数据并且从中获得有意义的见解。”

一些业界领先的企业和消费者 IT 公司已经在其产品和解决方案中利用了 MongoDB 的能力。MongoDB 3.0 版本引入了一种可插入式存储引擎以及 Ops Manager，它扩展了最适合 MongoDB 的应用程序集。

MongoDB 的名称源自“硕大(humungous)”一词。就像其他 NoSQL 数据库一样，MongoDB 也不遵循 RDBMS 原则。它没有表、行以及列的概念。另外，它也不提供 ACID 合规性、JOINS、外键等功能。

MongoDB 将数据存储为 Binary JSON 文档(也称为 BSON)。这些文档可以具有不同的模式，这意味着随着应用程序的演进，模式可以变更。MongoDB 是为可扩展性、性能和高可用性而构建的。

在本章中我们将探讨一些与 MongoDB 的创建以及设计决策有关的内容。我们将在后面几章中介绍 MongoDB 的关键功能、组件和架构。

3.1 历史

2007 年下半年，Dwight Merriman、Eliot Horowitz 及其团队决定开发一项在线服务。该服务的目的在于为开发、托管以及自动扩展的网络应用程序提供一个平台，这与 Google App Engine 或 Microsoft Azure 这样的产品是一致的。不久之后他们意识到没有开源的数据库平台能满足该服务的需求。

一年之后，用于该服务的这个数据库已经准备好使用了。该服务本身从来没有发布过，但这个团队于 2009 年决定将该数据库开源，其名称定为 MongoDB。2010 年 3 月，MongoDB 1.4.0 的发布被认为已经具备了产品化的特性。最新的正式版本是 3.0 并且已经于 2015 年 3 月发布。MongoDB 的构建受到了一家在纽约初创的公司 10gen 的资助。

3.2 MongoDB 设计原则

在 Eliot Horowitz 的其中一次谈话中，他提及，MongoDB 并非闭门造车式地设计出来的，而是基于构建大规模、高可用性以及强健系统的经验构建的。在本节中，我们将简要介绍使 MongoDB 发展成为今天这样的一些设计决策。

3.2.1 高速、可扩展性与敏捷性

在设计 MongoDB 时，设计团队的目标是创建一个快速、大规模可扩展并且易于使用的数据库。为了在分区的数据库中实现高速和横向可扩展性，正如 CAP 定理所阐释的，就必然有损一致性与事务支持度。因此，按照这个定理，MongoDB 在以一致性和事务支持度为代价的基础上提供了高可用性、可扩展性以及分区。实际上，这意味着相较于表和行，MongoDB 使用了文档以让其变得灵活、可扩展并且高速。

3.2.2 非关系型方法

传统的 RDBMS 平台使用了一种纵向扩展方法来提供可扩展性，这需要一台较快的服务器来提高性能。RDBMS 系统中的以下问题正是 MongoDB 以及其他 NoSQL 数据库应运而生的原因：

- 为了横向扩展，RDBMS 数据库需要链接存在于两个或多个系统中的数据以便反馈结果。这在 RDBMS 系统中是难以实现的，因为它们旨在处理所有的数据适用于共同计算的情况。因而这些数据必须适合在单一位置进行处理。
- 如果使用了多台双活(Active-Active)服务器，那么在这些服务器都从多个源得到更新时，判定哪个更新是正确的这一任务就会面临挑战。
- 当一个应用程序尝试从第二台服务器读取数据，并且该信息已经在第一台服务器上被更新但有待同步到第二台服务器时，那么所返回的信息可能就是过时的。

MongoDB 团队决定采用一种非关系型的方法来解决这些问题。正如所提到过的，MongoDB 将其数据存储在 BSON 文档中，其中所有的相关数据都被放置在一起，这意味着所有一切都在一个位置。MongoDB 中的查询是基于文档中的键的，因此这些文档可以分散到多台服务器。查询每台服务器意味着该服务器将检查其自己的文档集并且返回结果。这使得线性可扩展性和性能提升成为可能。

MongoDB 具有一种主从复制方式，其中主机器接受写请求。如果需要提升写的性能，则可以使用分片；这样就可以将数据划分到多台机器，并且让这些机器能够更新数据集的不同部分。分片在 MongoDB 中是自动化的；随着越来越多的机器被添加进来，数据就会被自动分布。

3.2.3 基于 JSON 的文档存储

MongoDB 使用了一种基于 JSON(JavaScript 对象标识法)的文档存储来存储数据。JSON/BSON 提供了一种无模式模型，这样就为数据库设计提供了灵活性。不同于在 RDBMS 中，对模式的变更可以无缝进行。

此设计还通过提供将相关数据在内部分组到一起，并且让其易于搜索的特性来获得高性能支持。

JSON 文档包含了实际的数据，并且相当于 SQL 中的行。不过，与 RDBMS 的行相反，文档可以具有动态模式。这意味着一个集合中的文档可以具有不同的字段或结构，或者通用的字段可以具有不同的数据类型。

一个文档包含了键-值对形式的数据。我们用一个示例来理解这一点：

```
{
"Name": "ABC",
"Phone": ["1111111",
........"222222"
........],
"Fax":..
}
```

正如所述，键和值都是成对出现的。文档中一个键的值可以留空。在上面的示例中，该文档具有三个键，分别是“Name”、“Phone”和“Fax”。“Fax”键没有值。

3.2.4 性能与功能对比

为了让 MongoDB 具有高性能并且快速运行，RDBMS 系统中常用的一些功能在 MongoDB 中都没有了。MongoDB 是一个面向文档的 DBMS，其中数据被存储为文档。它不支持 JOIN，并且它没有完全通用的事务。不过，它确实提供了对辅助索引的支持，这使得用户可以使用查询文档进行查询，并且它提供了对每个文档级别进行原子更新的支持。它提供了复制集，这是一种具有自动化故障转移的主从复制形式，并且它内置了横向扩展特性。

3.2.5 随处都能运行数据库

主要的一个设计决策就是随处运行数据库的能力，这意味着 MongoDB 应该能够运行在服务器、虚拟机或支持按需付费服务的云端。用于实现 MongoDB 的语言是 C++，这使得 MongoDB 能够达成这一目标。10gen 网站提供了用于不同操作系统平台的二进制，使得 MongoDB 可以运行在几乎所有类型的机器上。

3.3 与 SQL 的对比

下面是 MongoDB 不同于 SQL 的地方。

(1) MongoDB 使用文档来存储其数据，这就提供了一种灵活的模式(相同集合中的文档可以具有不同的字段)。这使得用户可以存储嵌套的或多值的字段，比如数组、散列等。相反，RDBMS 系统提供了一种固定模式，其中一个列的值应该具有类似的数据类型。另外，在一个单元格中存储数组或嵌套值也是不可能的。

(2) MongoDB 没有像 SQL 那样为 JOIN 操作提供支持。不过，它使得用户可以将所有相关数据一起存储到单个文档中，以避免在外围使用 JOIN。它提供了一种变通方法来解决这个问题。我们将在后面的章节中更为详尽地探讨这一点。

(3) MongoDB 没有像 SQL 那样提供对事务的支持。不过，它能确保文档级别的原子性。另外，它使用了一种隔离运算符来隔离影响多个文档的写操作，但它没有为多文档写操作提供“全有或全无”的原子性。

3.4 本章小结

在本章中，你对 MongoDB 有了初步了解，并且知道了 MongoDB 系统设计上的一些简要信息。在后面的章节中，将学习更多与 MongoDB 数据模型有关的知识。

第 4 章

MongoDB 数据模型

“MongoDB 被设计为在不需要任何预定义列或数据类型(不同于关系型数据库)的情况下处理文档，这使得其数据模型极具灵活性。”

在本章中，你将学习与 MongoDB 数据模型有关的知识。你还将了解灵活的模式(多态模式)意味着什么以及为何它是 MongoDB 数据模型的一个重要考量。

4.1 数据模型

在上一章中，你知道了 MongoDB 是一个基于文档的数据库系统，其中文档可以具有一种灵活的模式。这意味着一个集合中的文档可以具有不同(或相同)的字段集。这样就使得你在处理数据时拥有了更多的灵活性。

在本章中，你将探究 MongoDB 的灵活数据模型。在需要时，我们会采用与 RDBMS 系统对比的方式来揭示其差异。

MongoDB 部署可以使用许多数据库。每个数据库都是一组集合。集合类似于 SQL 中表的概念；不过，它们是无模式的。每个集合可以具有多个文档。可以将一个文档看作 SQL 中的一行。图 4-1 描述了 MongoDB 数据库模型。

在一个 RDBMS 系统中，由于表结构和每一列的数据类型都是固定的，因此你只能将特定数据类型的数据添加到一列中。在 MongoDB 中，一个集合是指一个文档的集合，其中数据被存储为键-值对。

我们用一个示例来理解数据是如何被存储到一个文档中的。以下文档保存了用户的姓名和手机号码：

```
{"Name": "ABC", "Phone": ["1111111", "222222" ] }
```

动态模式意味着相同集合中的文档可以具有相同或不同的字段集或结构，甚至通用字段都可以跨文档存储不同类型的值。在这里，数据被存储到集合文档中的方式并不死板。

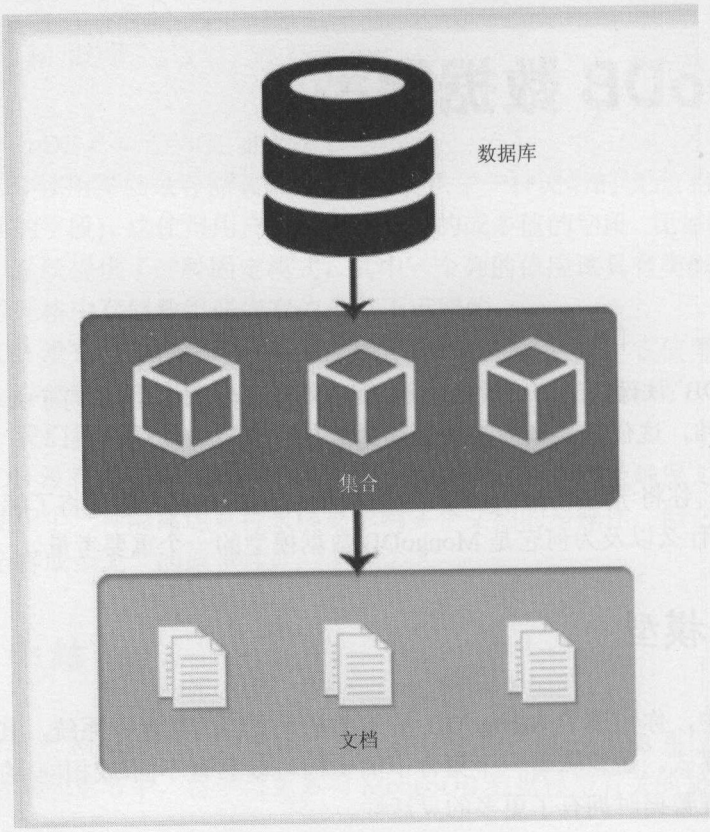

图 4-1 MongoDB 数据库模型

我们来看一个 Region 集合的示例：

```
{ "R_ID" : "REG001", "Name" : "United States" }
{ "R_ID" :1234, "Name" : "New York" , "Country" : "United States" }
```

在这段代码中，Region 集合中有两个文档。尽管这两个文档都是单个集合中的一部分，但它们具有不同的结构：第二个文档具有一个额外的信息字段，即 Country。实际上，如果你查看“R_ID”字段，就会发现在第一个文档中它存储了一个 STRING 值，而在第二个文档中它是一个数字。

因此，一个集合的文档可以具有完全不同的模式。将这些文档共同存储到某个特定集合或者使用多个集合来存储它们的任务是由应用程序完成的。

4.1.1 JSON 和 BSON

MongoDB 是一个基于文档的数据库。它使用二进制 JSON 来存储其数据。

在本节中，你将学习与 JSON 和二进制 JSON(Binary-JSON，BSON)有关的知识。JSON 指的是 JavaScript 对象表示法(JavaScript Object Notation)。它(与 XML 一起)是用于如今的现代网络数据交换的一种标准。其格式对于人类和机器来说都是可读的。它不仅

是交换数据的一种极佳方式，也是存储数据的一种极佳方式。

JSON 支持所有的基本数据类型(比如字符串、数字、布尔值以及数组)。

以下代码显示了一个 JSON 文档看起来的样子：

```
{
"_id" : 1,
"name" : { "first" : "John", "last" : "Doe" },
"publications" : [
            {
              "title" : "First Book",
              "year" : 1989,
              "publisher" : "publisher1"
            },
            { "title" : "Second Book",
              "year" : 1999,
              "publisher" : "publisher2"
            }
]
}
```

JSON 让你将所有相关的信息片段共同保存在一个位置，这就提供了极佳的性能。它还使得文档的更新变得独立。它是无模式的。

二进制 JSON(BSON)

MongoDB 会以二进制编码格式存储 JSON 文档，这被定义为 BSON。BSON 数据模型是 JSON 数据模型的一种扩展形式。

MongoDB 的 BSON 文档实现是快速、高度可遍历并且轻量级的。它支持将数组和对象嵌入到其他数组中，并且还使得 MongoDB 可以到达对象内部以便针对查询表达式构建索引和匹配对象，在顶层和嵌套的 BSON 键上都可以这样做。

4.1.2 标识符(_id)

MongoDB 将数据存储在文档中。文档是由键-值对组成的。尽管可以将一个文档比作 RDBMS 中的一行，但不同于一行的是，文档具有灵活的模式。一个键仅仅是一个标签而已，它差不多可以被比作 RDBMS 中的列名称。键用于从文档中查询数据。因此，就像一个 RDBMS 主键(用于唯一标识每一行)一样，你需要使用一个键唯一标识一个集合中的每个文档。这被称为 MongoDB 中的_id。

如果你没有为一个键显式指定任何值，那么将由 MongoDB 自动生成一个唯一值并且将这个值分配给这个键。这个键值是不可变更的，并且可以是除了数组外的任何数据类型。

4.1.3 固定集合

你现在非常精通集合与文档了。我们来探讨一种特殊类型的集合，它被称为固定集合。

MongoDB 有一个将集合固定的概念。这意味着它会按照插入的顺序在集合中存储文档。当集合达到其大小限制时，文档将会按照 FIFO(先进先出)顺序被从集合移除。这意味着最早被插入的文档将会首先被移除。

这对于需要自动维护插入顺序并且需要在超出固定大小后删除记录的使用场景来说是合适的。在超出特定大小后自动清空日志文件就是这样的一个使用场景。

提示：

MongoDB 本身将固定集合用于维护其复制日志。固定集合会确保按照插入顺序保存数据，因此按照插入顺序检索数据的查询会快速返回结果并且不需要索引。修改文档大小的更新是不允许的。

4.2 多态模式

由于你已经熟悉了 MongoDB 数据结构的无模式特性，因此我们现在来探究多态模式和使用场景。

多态模式就是集合具有不同类型或模式的文档的一种模式。这种模式的一个好例子就是名为 Users 的集合。一些用户文档可能具有一个额外的传真号码或者电子邮件地址，而其他的用户可能只有电话号码，但所有这些文档共同存在于相同的 Users 集合中。这一模式通常被称为多态模式。

在本章的这一部分，你将探究使用多态模式的各种缘由。

4.2.1 面向对象编程

面向对象编程使得你可以使用继承来让类共享数据和行为。它还让你可以在父类中定义能够在子类中重写的函数，因而将在不同的上下文中发挥不同的作用。换句话说，你可以使用相同的函数名称来像操作父类那样操作子类，尽管其背后的实现可能是不同的。这一特性被称为多态性。

在这种情况下，需要能够使用一种模式，其中一个层次结构中的所有相关对象集或对象都可以共同适应并且还可以同等地检索。

下面看一个示例。假定你有一个应用程序，它允许用户上传和共享不同的内容类型，比如 HTML 页面、文档、图片、视频等。尽管就上面提及的所有内容类型来说，其对应字段中的许多都很常见(比如姓名、ID、作者、上传日期以及时间)，但并非所有的字段都是相同的。例如，就图片而言，使用一个二进制字段来保存图片内容，而对于一个 HTML 页面，则要使用大型文本字段来保存 HTML 内容。

在这个场景中，可以使用 MongoDB 多态模式，其中所有的内容节点类型都被存储在相同的集合中，比如 LoadContent，并且每一个文档仅具有相应的字段。

```
// "Document collections" - "HTMLPage" document
{
id: 1,
title: "Hello",
type: "HTMLpage",
text: "<html>Hi..Welcome to my world</html>"
}
...
// Document collection also has a "Picture" document
{
id: 3,
title: "Family Photo",
type: "JPEG",
sizeInMB: 10,........
}
```

此模式不仅使得你可以在相同集合中用不同的结构将相关数据存储到一起，它还简化了查询。可以使用相同的集合来对常见字段执行查询，比如抓取于特定日期和时间上传的所有内容，也可以使用相同集合对特定字段进行查询，比如找出大小大于 X MB 的图片。

因此，面向对象编程是其中一个使用多态模式十分合理的场景。

4.2.2 模式演化

当你使用数据库时，需要考虑的其中一个最重要的注意事项就是模式演化(例如，在模式中变更对于运行中的应用程序的影响)。应该以对应用程序具有最小或没有影响的方式来完成其设计，这意味着不会停机或者最小化停机时间、没有代码变更或者代码变更非常少，等等。

典型的，在执行一个将数据库模式从旧版本更新为新版本的迁移脚本时就会发生模式演化。如果数据库不处于生产环境，那么该脚本就可以被简单删除并且重建数据库。然而，如果数据库处于生产环境并且包含实时数据，那么该迁移脚本就会很复杂，因为需要保存这些实时数据。该脚本应该将这一点纳入考量。尽管 MongoDB 提供了一个 Update 选项，当有一个新的字段要添加时，它可被用于更新一个集合中所有的文档结构。想象一下，如果你的集合中有上千个文档需要这样做，会产生怎样的影响。其过程将非常缓慢并且对于基础应用程序的性能会产生不良影响。完成这项任务的其中一种方式就是将新的结构包含在新的要添加到集合的文档中，然后当应用程序仍旧处于运行状态时逐步在后台迁移该集合。这就是使用多态模式将带来好处的许多使用场景中的一个。

例如，假设你正在使用一个 Tickets 集合，其中有具有票据详情的文档，如下所示：

```
// "Ticket1" document (stored in "Tickets" collection")
{
```

```
id: 1,
Priority: "High",
type: "Incident",
text: "Printer not working"
}...........
```

在某些时候，应用程序团队决定在票据文档结构中引入一个“简要描述”字段，因此最佳备选方案就是在新的票据文档中引入这一新字段。在该应用程序内部，你嵌入了一段代码，它将处理对“旧版本”文档(不具有简要描述字段)以及“新版本”文档(具有简要描述字段)的检索。旧版本文档可以逐渐被迁移到新版本文档。一旦迁移完成，如果需要，则可以更新代码来移除该段被嵌入以处理缺失字段的代码。

4.3 本章小结

在本章中，你学习了与 MongoDB 数据模型有关的知识，你还了解了标识符和固定集合。在本章最后，你理解了灵活模式会起到多大的作用。

在下一章中，你将开始使用 MongoDB。你要执行 MongoDB 的安装和配置。

第 5 章

MongoDB-安装与配置

“MongoDB 是一种跨平台数据库。”

在本章中，你将仔细了解在 Windows 和 Linux 上安装 MongoDB 的过程。

5.1 选择你的版本

MongoDB 可以运行在大多数平台上。MongoDB 下载页面 www.mongodb.org/downloads 上列出了可用的所有安装包。

适合于你环境的正确版本取决于服务器的操作系统以及处理器类型。MongoDB 同时支持 32 位和 64 位架构，但推荐在你的生产环境中使用 64 位版本。

---

32 位版本的限制：这是由于 MongoDB 中使用了内存映射文件造成的。这就将 32 位版本限制为约 2GB 数据。出于性能因素的考量，推荐将 64 位版本用于生产环境。

在编写本书时，最新的 MongoDB 正式版本为 3.0.4，提供了可用于 Linux、Windows、Solaris 以及 Mac OS X 的 MongoDB 下载。

MongoDB 下载页面被划分成了以下几部分：

- 当前的稳定版本(3.0.4)–6/16/2015
- 先前版本(稳定的)
- 开发版本(不稳定的)

当前版本是可用的最稳定的最新版本，在编写本书时的最新版本为 3.0.4。当一个新版本被发布时，之前的稳定版本会被移到先前版本部分。

顾名思义，开发版本是仍处于开发阶段的版本，因而被标记为不稳定。这些版本可能具有额外的功能，但它们可能是不稳定的，因为它们仍处于开发阶段。可以使用开发版本来尝试新的功能并且就功能和面临的问题向 10gen 提供反馈。

5.2 在 Linux 上安装 MongoDB

本节将介绍 Linux 系统上 MongoDB 的安装。为了后续的阐释，我们将使用一种

Ubuntu Linux 发行版。可以手动或通过仓储安装 MongoDB。我们将介绍这两种选项。

5.2.1 使用仓储进行安装

在 Linux 中，仓储就是包含软件的在线目录。Aptitude 是用于在 Ubuntu 上安装软件的程序。尽管默认仓储中可能已经提供了 MongoDB，但它可能是过时的版本，因此第一步是配置 Aptitude 来查看自定义仓储。

(1) 执行以下命令为 MongoDB 导入 public.GPG 键：

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
7F0CEB10
```

(2) 接下来，使用以下命令创建/etc/apt/sources.list.d/mongodb-org-3.0.list 文件：

```
echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/
mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/
mongodb-org-3.0.list
```

(3) 最后，使用以下命令重载该仓储：

```
sudo apt-get update
```

现在 Aptitude 就知道手动添加的仓储了。

(4) 接下来，需要安装该软件。应该在 shell 中执行以下命令，以便安装 MongoDB 的当前稳定版本：

```
sudo apt-get install -y mongodb-org
```

你已经成功安装了 MongoDB，这就是全部的安装步骤。

5.2.2 手动安装

在本节中，将看到如何手动安装 MongoDB。这一知识在以下情况下会很重要：

- 当 Linux 发行版不使用 Aptitude 时。
- 当需要的版本无法通过仓储获得或者不是仓储的一部分时。
- 当需要同时运行多个 MongoDB 版本时。

手动安装的第一步是决定使用哪个 MongoDB 版本，然后从网站上下载它。接着，需要使用以下命令来提取安装包：

```
$ tar -xvf mongodb-linux-x86_64-3.0.4.tgz
mongodb-linux-i686-3.0.4/THIRD-PARTY-NOTICES
mongodb-linux-i686-3.0.4/GNU-AGPL-3.0
mongodb-linux-i686-3.0.4/bin/mongodump
.............
mongodb-linux-i686-3.0.4/bin/mongosniff
```

```
mongodb-linux-i686-3.0.4/bin/mongod
mongodb-linux-i686-3.0.4/bin/mongos
mongodb-linux-i686-3.0.4/bin/mongo
```

这样就能将安装包内容提取到一个新的目录，其名称为 mongodb-linux-x86_64-3.0.4(它位于你的当前目录下)。该目录包含许多子目录和文件。主要的可执行文件位于子目录 bin 中。

这样就成功完成了 MongoDB 的安装。

5.3 在 Windows 上安装 MongoDB

在 Windows 上安装 MongoDB 很简单，只需要下载用于所选 Windows 版本的 msi 文件并且运行安装程序即可。

安装程序将指导你完成 MongoDB 的安装。

跟随该向导，你将看到 Choose Setup Type 界面。有两个可用的安装类型，其中可以自定义你的安装。在这个示例中，选择 Custom 安装类型。

需要在选择 Custom 时指定安装目录，因此将该目录指定为 C:\PracticalMongoDB。

注意，MongoDB 可以从用户所选择的任意文件夹运行，因为它是自包含的并且对系统没有依赖。如果选择了 Complete 安装类型，那么默认选择的文件夹就是 C:\Program Files\MongoDB。

单击 Next 按钮会将你带到 Ready to installation 界面。单击 Install 按钮。

这样就会开始安装并且将在界面上显示安装进度。一旦安装完成，该向导会向你显示完成界面。

单击 Finish 就会完成安装。在上述步骤成功完成之后，你就有了一个名称为 C:\PracticalMongoDB 的目录，其中所有相关的应用程序都位于 bin 文件夹中。这就是全部的安装步骤了。

5.4 运行 MongoDB

我们来看看如何开始运行和使用 MongoDB。

5.4.1 先决条件

需要一个数据文件夹用于存储文件。默认情况下，在 Windows 中这个文件夹是 C:\data\db，而在 Linux 系统中是/data/db。

这些数据目录并非由 MongoDB 创建，因此在打开 MongoDB 之前，需要手动创建该数据目录，并且你需要确保设置了合适的权限(比如 MongoDB 具有读取、写入以及目录创建权限)。

如果在你创建该文件夹之前打开 MongoDB，那么它会抛出一条错误消息并且将运行

失败。

5.4.2 开启服务

一旦目录被创建并且设置好了权限，就可以执行 mongod 应用程序(放置在 bin 目录下)来开启 MongoDB 核心数据库服务。

作为上述安装的延续，可以通过在 Windows 中打开命令提示符(需要以管理员身份运行)并且执行以下命令来开启相同的服务：

```
c:\> c:\practicalmongodb\bin\mongod.exe
```

在 Linux 环境下，mongod 程序是在 shell 中开启的。

这将在本地主机接口上开启 MongoDB 数据库。它将侦听端口 27017 上来自 mongo shell 的连接。

正如所提到过的，需要在开启数据库之前创建文件夹路径，默认路径是 c:\data\db。在开启数据库服务时也可以通过使用–dbpath 参数来提供另一个路径。

```
C :\> C:\ practicalmongodb \bin\mongod.exe --dbpath
C:\NewDBPath\DBContents
```

5.5 验证安装结果

子目录 bin 下提供了相关的可执行程序。可以在 bin 目录下检查以下程序以便审查安装步骤是否成功：

- Mongod：核心数据库服务器
- Mongo：数据库 shell
- Mongos：自动分片程序
- Mongoexport：导出实用程序
- Mongoimport：导入实用程序

除了上述程序之外，bin 文件夹中还提供了其他的应用程序。

mongo 应用程序会启动 mongo shell，它提供了对数据库内容的访问并且可以对 MongoDB 中的数据发起选择性查询或者执行聚合。

正如你在上面看到的，mongod 应用程序用于开启数据库服务或者守护程序。

在启动应用程序时，可以设置多个标记。例如，–dbpath 可用于指定一个替代路径，用于指定数据库文件应该存储的位置。要得到所有可用的选项列表，可以在启动服务时包含--help 标记。

5.6 MongoDB Shell

mongo shell 是作为 MongoDB 标准发行版本的一部分来提供的。这个 shell 为

MongoDB 提供了完整的数据库接口，让你可以使用一个 JavaScript 环境来处理 MongoDB 中存储的数据，该环境支持对 JavaScript 语言和所有标准函数的完全访问。

一旦开启了数据库服务，你就可以启动 mongo shell 并且开始使用 MongoDB。可以使用 Linux 中的 Shell 或者 Windows 中的命令提示符(以管理员身份运行)来完成这一任务。

你必须引用可执行文件的提取位置，比如 Windows 环境中的 C:\practicalmongodb\bin\ 文件夹。

打开命令提示符(以管理员身份运行)并且输入 mongo.exe。按下 Enter 键，这样就会开启 mongo shell。

```
C:\> C:\practicalmongodb\bin\mongo.exe
MongoDB shell version: 3.0.4
connecting to: test
>
```

如果在开启该服务时没有指定参数，那么它将连接到本地主机实例上名称为 test 的默认数据库。

当连接到该数据库时，它会被自动创建。MongoDB 提供了在尝试访问一个不存在的数据库时自动创建该数据库的功能。

下一章将提供与使用 mongo shell 有关的更多信息。

5.7 保障部署安全

经由默认配置，你知道了如何安装和开始使用 MongoDB。接下来，你需要确保数据库中存储的数据从各方面来说都是安全的。

在本节中，你将了解如何确保你的数据安全。你将修改默认安装配置来确保你的数据库更加安全。

5.7.1 使用身份验证和授权

身份验证会验证用户的身份，而授权将判定用户可以在经过身份验证的数据库上执行的操作级别。

这意味着用户仅在使用能够访问数据库的凭证登录之后才能访问数据库。这样一来就禁止了对数据库的匿名访问。在用户通过验证之后，就可以使用授权来确保用户仅拥有需要用来完成手头任务所需的访问量。

身份验证和授权都存在于单数据库级别。用户存在于单个逻辑数据库的上下文中。

关于用户的信息被保存在一个名称为 system.users 的集合中，该集合存在于管理数据库中。这个集合会保存对用户进行身份验证所需的凭据，其中存储了用户 id、密码和创建该集合所面向的数据库，外加用于对用户授权所需的权限。

MongoDB 使用了一种基于角色的方式用于授权(read、readWrite、readAnyDatabase

等角色)。如果需要，用户管理员可以创建自定义角色。

system.users 集合中的一个权限文档被用于存储每个用户角色。相同的文档也会保存经过身份验证的用户的凭据。

以下是 system.users 集合中一个文档的示例：

```
{
_id : "practicaldb.Shaks",
user : "Shaks",
db : "practicaldb",
credentials : {.......},
roles : [
{ role: "read", db: "practicaldb" },
{ role: "readWrite", db: "MyDB" }
],
......
}
```

这个文档告诉我们，用户 Shaks 被关联到了数据库 practicaldb，并且它在 practicaldb 数据库中具有 read 角色，而在 MyDB 数据库中具有 readWrite 角色。注意，用户名称和所关联的数据库唯一标识了 MongoDB 中的一个用户，因此如果你有两个用户具有相同的名称，但它们关联到了不同的数据库，那么它们就会被认为是两个不同的用户。因此，一个用户可以在不同的数据库中拥有具有不同授权级别的多个角色。

可用的角色有：

- read：这个角色提供了对指定数据库所有集合的只读访问。
- readWrite：这个角色提供了指定数据库中对任意集合的读写访问。
- dbAdmin：这个角色使得用户可以在指定数据库中执行管理操作，比如使用 ensureIndex、dropIndexes、reIndex、indexStats 管理索引、重命名集合、创建集合等。
- userAdmin：这个角色使得用户可以对指定数据库的 system.users 集合执行 readWrite 操作。它还启用了对已有用户的权限进行修改或者创建新用户的功能。这实际上是指定数据库的超级用户角色。
- clusterAdmin：这个角色使得用户可以对修改或显示与整个系统有关的信息的管理操作授予访问权限。clusterAdmin 只适用于管理数据库。
- readAnyDatabase：这个角色使得用户可以读取 MongoDB 环境中的任意数据库。
- readWriteAnyDatabase：这个角色类似于 readWrite，只不过它适用于所有数据库。
- userAdminAnyDatabase：这个角色类似于 userAdmin 角色，只不过它适用于所有数据库。
- dbAdminAnyDatabase：这个角色与 dbAdmin 相同，只不过它适用于所有数据库。
- 从 2.6 版本开始，一个用户管理员还可以通过提供集合级别以及命令级别的访问权限来创建遵循最小权限策略的用户定义的角色。用户定义的角色仅作用于创建

它的数据库中，并且被数据库和角色名称的组合唯一标识。所有的用户定义角色都被存储在 system.roles 集合中。

1. 启用身份验证

身份验证默认是禁用的，因此要使用--auth 来启用身份验证。当开启 mongod 时，则要使用 mongod --auth。在启用身份验证之前，你需要至少一个管理用户。正如你在上面所看到的，一个管理用户就是负责创建和管理其他用户的用户。

建议在生产部署中仅仅为了管理用户才创建这样的用户，并且不应该将其用于任何其他角色。在 MongoDB 部署中，这个用户是需要被创建的第一个用户；系统的其他用户可以由这个用户来创建。

有两种方式可以创建管理用户：在启用身份验证之前或者在启用身份验证之后。

在这个示例中，你首先将创建管理用户，然后启用身份验证设置。以下步骤应该在 Windows 平台上执行。

使用默认设置开启 mongod：

```
C:\>C:\practicalmongodb\bin\mongod.exe
C:\practicalmongodb\bin\mongod.exe --help for help and startup options

2015-07-03T23:11:10.716-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero out data files
2015-07-03T23:11:10.716-0700 I JOURNAL [initandlisten] journal dir=C:\
data\db\journal

.....................................................

2015-07-03T23:11:10.763-0700 I CONTROL [initandlisten] MongoDBstarting
:pid=2776port=27017 dbpath=C:\data\db\ 64-bit host=ANOC9
2015-07-03T23:11:10.763-0700 I CONTROL [initandlisten] targetMinOS:
Windows 7/Windows Server 2008 R2
2015-07-03T23:11:10.763-0700 I CONTROL [initandlisten] db version v3.0.4
2015-07-03T23:11:10.764-0700 I CONTROL [initandlisten] OpenSSL version:
OpenSSL 1.0.1j-fips 19 Mar 2015
2015-07-03T23:11:10.764-0700 I CONTROL [initandlisten] build info:
windows sys.getwindowsversion(major=6, minor=1, build=7601, platform=2,
service_pack='Service Pack 1')
BOOST_LIB_VERSION=1_49
2015-07-03T23:11:10.771-0700 I NETWORK [initandlisten] waiting for
connections on port 27017
```

2. 创建管理用户

以管理员身份运行另一个命令提示符实例，并且执行 mongo 应用程序：

```
C:\> C:\practicalmongodb\bin\mongo.exe
MongoDB shell version: 3.0.4
connecting to: test
>
```

切换到管理数据库

注意，admin 数据库是一个具有特别权限的数据库，用户需要访问它以便执行某些管理命令，比如创建一个 admin 用户。

```
>db = db.getSiblingDB('admin')
```

管理员

需要为用户创建这两个角色中的一个：userAdminAnyDatabase 或 userAdmin：

```
>db.createUser({user: "AdminUser", pwd: "password", roles:["userAdmin-
AnyDatabase"]})
Successfully added user: { "user" : "AdminUser", "roles" : [ "userAdmin-
AnyDatabase" ] }
```

接下来，使用这个用户进行身份验证。使用 auth 设置重启 mongod：

```
C:\> C:\practicalmongodb\bin\mongod.exe -auth
C:\practicalmongodb\bin\mongod.exe --help for help and startup options
2015-07-03T23:11:10.716-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero out data files
2015-07-03T23:11:10.716-0700 I JOURNAL [initandlisten] journal
dir=C:\data\db\journal
.....................................................
2015-07-03T23:11:10.763-0700 I CONTROL [initandlisten] MongoDBstarting
:pid=2776
port=27017 dbpath=C:\data\db\ 64-bit host=ANOC9
2015-07-03T23:11:10.763-0700 I CONTROL [initandlisten] targetMinOS:
Windows 7/Windows Server 2008 R2
2015-07-03T23:11:10.763-0700 I CONTROL [initandlisten] db version v3.0.4
2015-07-03T23:11:10.764-0700 I CONTROL [initandlisten] OpenSSL version:
OpenSSL 1.0.1j-fips 19 Mar 2015
2015-07-03T23:11:10.764-0700 I CONTROL [initandlisten] build info:
windows sys.
getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack=
'Service Pack 1')
BOOST_LIB_VERSION=1_49
```

```
2015-07-03T23:11:10.771-0700 I NETWORK [initandlisten] waiting for
connections on port 27017
```

使用上面创建的 AdminUser 用户对 admin 数据库启动 mongo 控制台和身份验证：

```
C:\>c:\practicalmongodb\bin\mongo.exe
MongoDB shell version: 3.0.4
connecting to: test
>use admin
switched to db admin
>db.auth("AdminUser", "password")
1
>
```

3. 创建一个用户并启用授权

在本节中，你将创建一个用户并且为这一新创建的用户分配一个角色。你已经使用 admin 用户进行了身份验证，如下所示：

```
C:\>c:\practicalmongodb\bin\mongo.exe
MongoDB shell version: 3.0.4
connecting to: test
>use admin
switched to db admin
>db.auth("AdminUser", "password")
1
>
```

切换到 Product 数据库，并且创建用户 Alice 以及分配对 product 数据库的读取访问权限，就像这样：

```
> use product
switched to db product
>db.createUser({user: "Alice"
... , pwd:"Moon1234"
... , roles: ["read"]
... }
... )
Successfully added user: { "user" : "Alice", "roles" : [ "read" ] }
```

接下来，验证该用户是否已经具有对该数据库的只读访问权限：

```
>db
product
>show users
{
        "_id" : "product.Alice",
        "user" : "Alice",
```

```
        "db" : "product",
        "roles" : [
                {
                        "role" : "read",
                "db" : "product"
                }
        ]
}
```

接下来，连接到一个新的 mongo 控制台并且用 Alice 登录到 Products 数据库以运行只读命令：

```
C:\> c:\practicalmongodb\bin\mongo.exe -u Alice -p Moon1234 product
2015-07-03T23:11:10.716-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero-out data files
MongoDB shell version: 3.0.4
connecting to: products

Post successful authentication the following entry will be seen on the
mongod console.

2015-07-03T23:11:26.742-0700 I ACCESS [conn2] Successfully authenticated
as principal Alice on product
```

5.7.2 控制网络访问

默认情况下，mongod 和 mongos 会绑定到一个系统上所有可用的 IP 地址。在本节中，你将了解用于限制网络访问的配置选项。下面的代码是在 Windows 平台上执行的：

```
C:\> c:\practicalmongodb\bin\mongod.exe --bind_ip 127.0.0.1 --port 27017
--rest
2015-07-03T00:33:49.929-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero out data files
2015-07-03T00:33:49.946-0700 I JOURNAL [initandlisten] journal dir=C:\
data\db\journal
2015-07-03T00:33:49.980-0700 I CONTROL [initandlisten] MongoDBstarting
:pid=1144 port=27017 dbpath=C:\data\db\ 64-bit host=ANOC9
2015-07-03T00:33:49.980-0700 I CONTROL [initandlisten] targetMinOS:
Windows 7/Windows Server 2008 R2
2015-07-03T00:33:49.980-0700 I CONTROL [initandlisten] db version v3.0.4
2015-07-03T00:33:49.980-0700 I CONTROL [initandlisten] OpenSSL version:
OpenSSL1.0.1j-fips 19 Mar 2015
2015-07-03T00:33:49.980-0700 I CONTROL [initandlisten] build info:
windows sys.getwindowsversion(major=6, minor=1, build=7601, platform=2,
service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
```

```
2015-07-03T00:33:49.981-0700 I CONTROL [initandlisten] allocator: system
2015-07-03T00:33:49.981-0700 I CONTROL [initandlisten] options: { net:
{ bindIp: "127.0.0.1", http: { RESTInterfaceEnabled: true, enabled: true },
port: 27017} }
2015-07-03T00:33:49.990-0700 I NETWORK [initandlisten] waiting for
connections on port 27017
2015-07-03T00:33:49.990-0700 I NETWORK [websvr] admin web console waiting
for connections on port 28017
2015-07-03T00:34:22.277-0700 I NETWORK [initandlisten] connection
accepted from 127.0.0.1:49164 #1 (1 connection now open)
```

你已经使用 bind_ip 启动了服务器，它只有一个设置为 127.0.0.1 的值，这个值是本地主机的接口。

bind_ip 会限制入站连接的网络接口，程序会侦听这些接口。可以指定逗号分隔的 IP 地址。在你的例子中，已经将 mongod 限制为仅侦听本地主机接口。

当 mongod 实例启动时，默认情况下它会等待所有在端口 27017 上的入站连接。可以使用-port 修改此设置。

仅修改端口并不会降低太多风险。为了完全保障环境的安全，你需要使用防火墙设置来仅允许受信任的客户端连接该端口。

修改这个端口还会变更 HTTP 状态接口端口，默认情况下该端口是 28017。这个端口在 X+1000 的端口上是可用的，其中 X 表示连接端口。

这个网页公开了诊断和监控信息，其中包括操作数据、各种日志以及关于数据库实例的状态报告。它提供了管理级别的统计信息，可用于管理运营。这个页面默认情况下是只读的；要让它完全可交互，你要使用 REST 设置。这一配置会让该页面完全可交互，有助于管理员排除任何性能问题。应该使用防火墙仅允许受信任的客户端访问此端口。

建议禁用生产环境中的 HTTP 状态页面和 REST 配置。

1. 使用防火墙

防火墙被用于控制网络内部的访问。它们可用于允许从特定 IP 地址到特定 IP 端口的访问权限，或者停止来自任何不受信任的主机的访问。它们可用于为你的 mongod 实例创建一个受信任的环境，其中可以指定哪些 IP 地址或主机可以连接到 mongod 的哪些端口或接口。

在 Windows 平台上，使用 netsh 为端口 27017 配置入站通信：

```
C:\>netshadvfirewall firewall add rule name="Open mongod port 27017"
dir=in action=allow protocol=TCP localport=27017
Ok.
C:\>
```

这段代码表明，所有的入站通信在端口 27017 上都是允许的，因此任何应用程序服务器都可以连接到 mongod。

2. 数据加密

你已经看到了，MongoDB 会将其全部数据存储在一个数据目录中，在 Windows 中，这个目录默认为 C:\data\db，而在 Linux 中为/data/db。这些文件在该目录中是未加密存储的，因为 Mongo 中没有提供自动加密这些文件的方法。任何具有文件系统访问权限的攻击者都可以读取这些文件中存储的数据。确保敏感信息在被写入数据库之前加密是应用程序的职责。

此外，应该实现像文件系统级别加密和权限许可这样的操作系统级别机制以便阻止对这些文件的未授权访问。

3. 通信加密

对 mongod 和客户端(比如 mongo shell)之间的通信进行加密是一个常见的需求。在此设置中，你将看到如何通过配置 SSL 来将一种更高级别的安全控制添加到上面的安装中，以便 mongod 和 mongo shell(客户端)之间的通信使用一个 SSL 证书和密钥来进行。

建议将 SSL 用于服务器和客户端之间的通信。

从版本 3.0 开始，目前大多数 MongoDB 发行版本都已包含对 SSL 的支持。以下命令是在 Windows 平台上执行的。

第一步是生成包含公钥证书和私钥证书的.pem 文件。MongoDB 可以使用自签名证书或者任意由证书颁发机构发布的有效证书。

在本书中，你将使用以下命令生成一个自签名的证书和私有密钥。

(1) 根据 MongoDB 的发行版本以及 Windows 平台版本安装 OpenSSL 和 Microsoft Visual C++ 2008 可再发行组件。在本书中，你已经安装了 64 位版本。

(2) 运行以下命令创建一个公钥证书和一个私有密钥：

```
C:\> cd c:\OpenSSL-Win64\bin
C:\OpenSSL-Win64\bin\>openssl
```

这样就能打开 OpenSSL shell，在其中你需要输入以下命令：

```
OpenSSL>req -new -x509 -days 365 -nodes -out C:\practicalmongodb\
mongodb-cert.crt -keyout C:\practicalmongodb\mongodb-cert.key
```

以下步骤会生成一个名称为 mongodb-cert.key 的证书密钥并将它放置在 C:\practicalmongodb 文件夹中。

(3) 接下来，需要将该证书和私有密钥与.pem 文件连接起来。为此，请在命令提示

符下运行以下命令：

```
C:\> more C:\practicalmongodb\mongodb-cert.key > temp
C:\> copy \b temp C:\practicalmongodb\mongodb-cert.crt > C:\practicalmongodb\
mongodb.pem
```

现在你有了一个.pem 文件。在开启 mongod 时使用以下运行时选项：

```
C:\> C:\practicalmongodb\bin\mongod -sslMode requireSSL --sslPEMKeyFile
C:\practicalmongodb\mongodb.pem
2015-07-03T03:45:33.248-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero-out data files
2015-07-03T02:54:30.630-0700 I JOURNAL [initandlisten] journal
dir=C:\data\db\journal
2015-07-03T02:54:30.670-0700 I CONTROL [initandlisten] MongoDBstarting
:pid=2816 port=27017 dbpath=C:\data\db\ 64-bit host=ANOC9
2015-07-03T02:54:30.670-0700 I CONTROL [initandlisten] targetMinOS:
Windows 7/Windows Server 2008 R2
2015-07-03T02:54:30.670-0700 I CONTROL [initandlisten] db version v3.0.4
2015-07-03T02:54:30.670-0700 I CONTROL [initandlisten] OpenSSL version:
OpenSSL1.0.1j-fips 19 Mar 2015
2015-07-03T02:54:30.670-0700 I CONTROL [initandlisten] build info:
windows sys.getwindowsversion(major=6, minor=1, build=7601, platform=2,
service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
2015-07-03T02:54:30.671-0700 I CONTROL [initandlisten] allocator: system
2015-07-03T02:54:30.671-0700 I CONTROL [initandlisten] options: { net:
{ ssl: {PEMKeyFile: "c:\practicalmongodb\mongodb.pem", mode: "requireSSL" } } }
2015-07-03T02:54:30.680-0700 I NETWORK [initandlisten] waiting for
connections on port 27017 ssl
2015-07-03T03:33:43.708-0700 I NETWORK [initandlisten] connection
accepted from 127.0.0.1:49194 #2 (1 connection now open)
```

提示：

不建议在生产环境中使用自签名证书，除非处于受信任的网络中，因为该证书将让你容易受到中间人攻击。

接下来将使用 mongo shell 连接到上面的 mongod。当使用--ssl 选项运行 mongo 时，需要指定--sslAllowInvalidCertificates 或--sslCAFile。我们使用--sslAllowInvalidCertificates。

打开一个终端窗口并且输入以下命令：

```
C:\> C:\practicalmongodb\bin>mongo --ssl --sslAllowInvalidCertificates
2015-07-03T02:30:10.774-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero-out data files
MongoDB shell version: 3.0.4
connecting to: test
```

5.8 使用 MongoDB 云管理器进行配置

在本章开头学习了如何使用 Windows 和 Linux 安装与配置 MongoDB。在本章的这一部分将了解如何使用 MongoDB 云管理器。

MongoDB 云管理器是由数据库开发人员内置的一个监控解决方案。在版本 2.6 之前，MongoDB 云管理器(之前被称为 MongoDB 监控服务或 MMS)仅被用于监控和管理 MongoDB。从版本 2.6 开始，重要的增强功能被引入到了 MongoDB 云管理器，其中包括备份、时点恢复以及一种让操作 MongoDB 的任务比之前更简单的自动化操作功能。该自动化操作功能为管理员提供了几次单击就能快速创建、升级、扩展或关闭 MongoDB 实例的强大能力。

在本书的这一部分，将看到如何开始使用 MongoDB 云管理器。你将使用 MongoDB 云管理器在 AWS 上部署一个独立的 MongoDB 实例。

当开始使用 MongoDB 云管理器时，它会要求在每一台服务器上安装一个自动化操作代理，然后该代理会被 MongoDB 云管理器用来与服务器通信。

为开始配置，首先需要在 MongoDB 云管理器上创建你的档案。

输入这个 URL：https://cloud.mongodb.com。单击 Login 或 Sign up for Free 按钮，这取决于你是否具有一个账户。

由于你是第一次使用，因此要单击 Sign up for Free 按钮。这样就会显示图 5-1 中描述的页面。

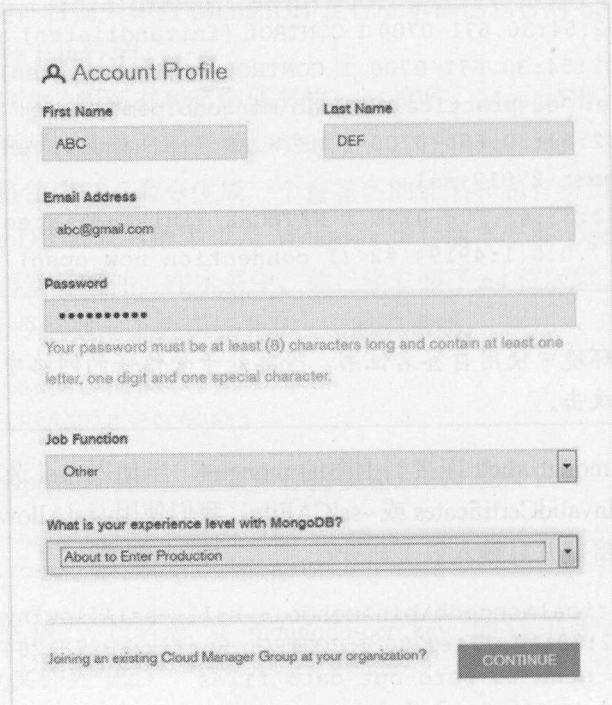

图 5-1 账户档案

你将创建一个新的账户档案。不过，MongoDB 提供了一个选项用于作为已有的云管理器组的联结。

输入所有相关的详细信息，如图 5-1 所示，并且单击 Continue 按钮。这样就会显示提供公司信息的页面。一旦你完成了账户档案和公司信息的输入，则可以接受条款并且单击 Create Account 按钮。这样就完成了账户档案的创建。下一个步骤是创建一个分组(见图 5-2)。

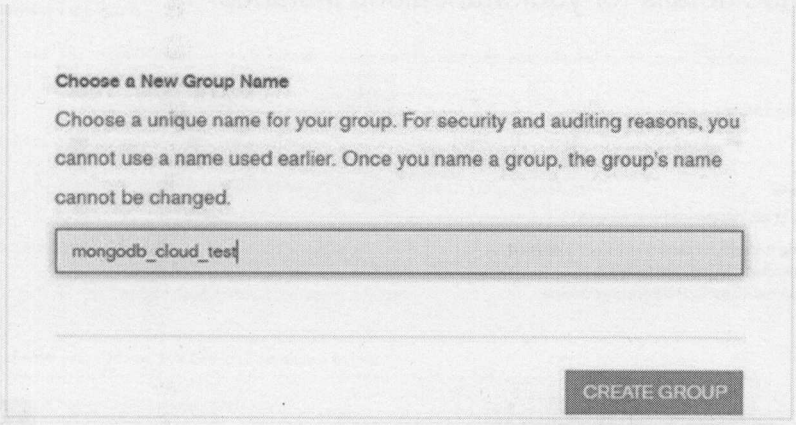

图 5-2 创建分组

为该分组提供一个唯一名称，并且单击 Create Group 按钮。接下来是图 5-3 中所示的部署选择页面，其中可以选择构建一个新的部署或者管理一个现有部署。

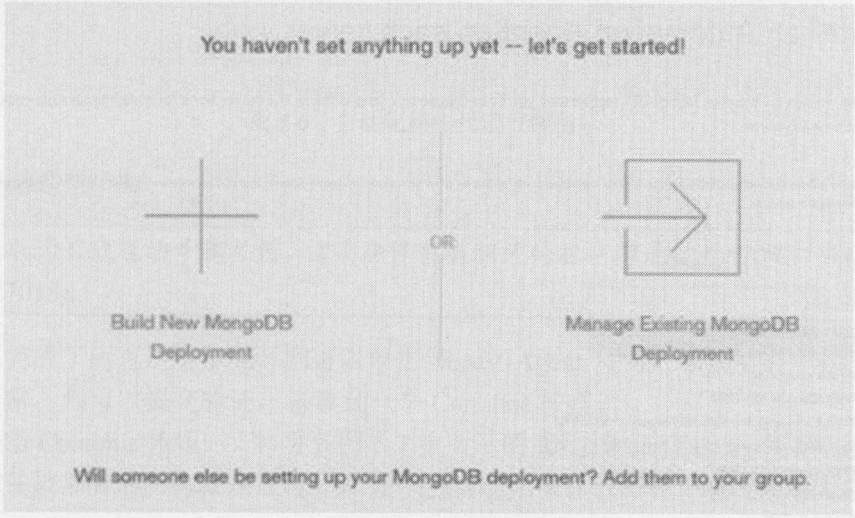

图 5-3 部署

选择构建一个新的部署。接着，将提示你构建该部署的位置(例如，本地、AWS 或者其他远程环境)。在这个示例中，选择 AWS。单击 Deploy in AWS 选项将让你在自主配置和使用云管理器进行配置之间选择。

选择“I will Provision”选项，这意味着你要使用一台 AWS 上已经为你配置好的机器。

下一个界面提供了用于部署类型的选项(例如，独立、副本集或分片群集)。你正在进行独立部署，因此单击 Create Standalone 选框。这样就会向你显示图 5-4 中所示的界面。

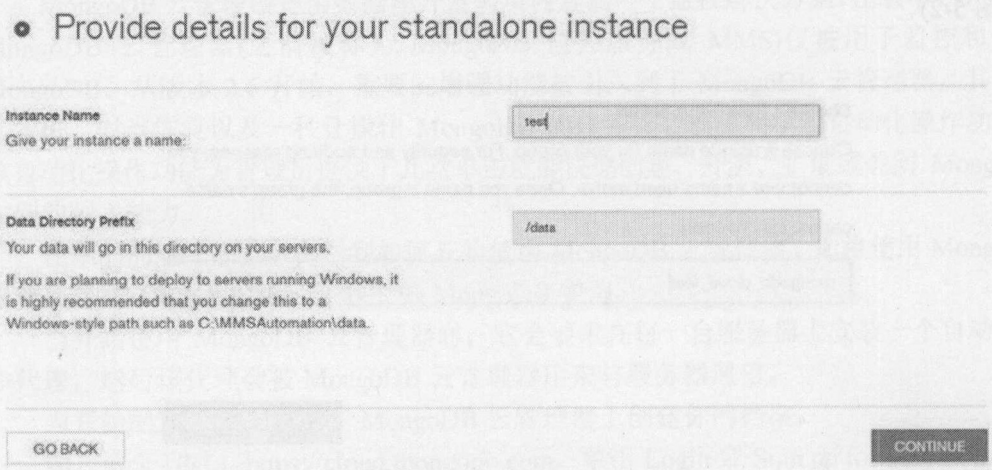

图 5-4 一个独立实例的详细信息

提供实例名称和数据目录前缀，并且单击 Continue 按钮。接下来是图 5-5 中所示的界面，它会提示你在每台服务器上安装一个自动化操作代理。

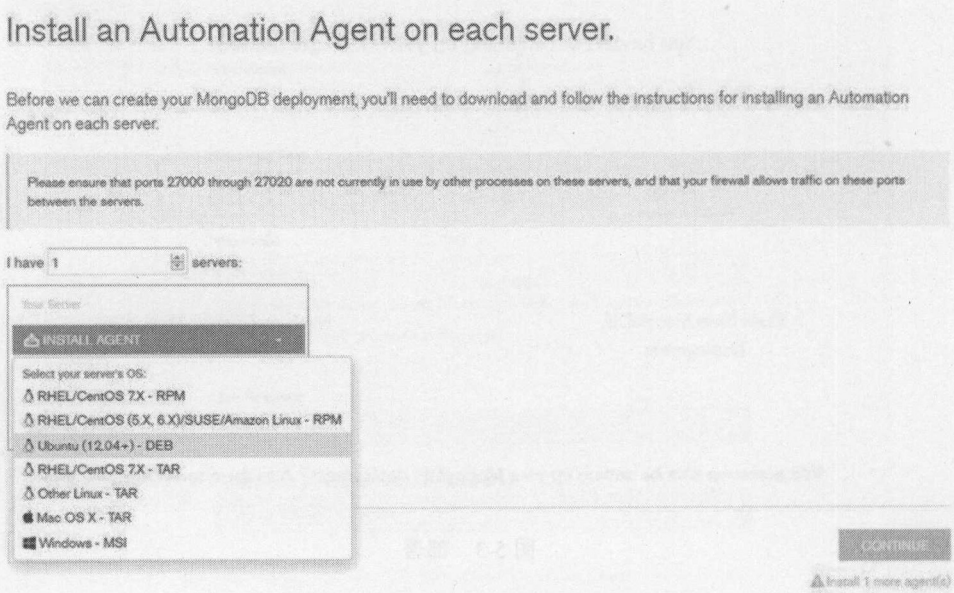

图 5-5 安装一个自动化操作代理

这个界面提供了指定服务器数量的选项。在这个示例中，要指定该数量为 1。接下来，需要指定平台。选择 Ubuntu。然后会出现图 5-6 中的界面。

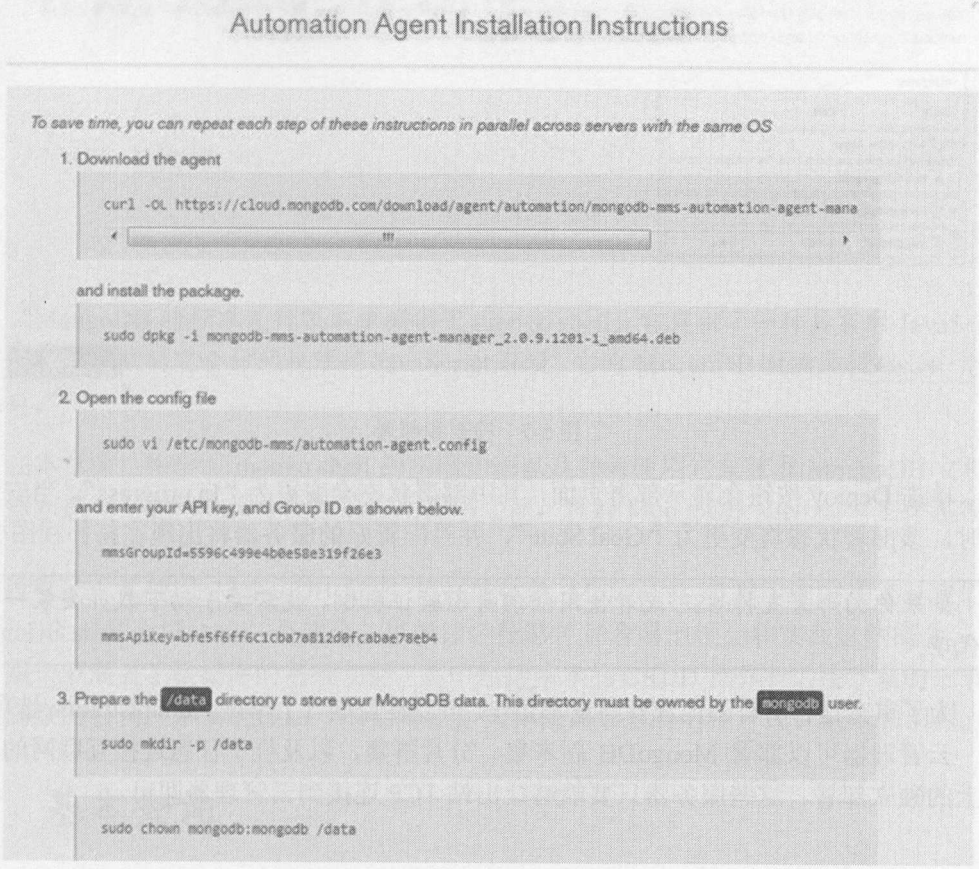

图 5-6 自动化操作代理安装指示

请遵循这些步骤。

在完成开启代理的步骤之前，需要确保所有相关的端口都是打开的(端口 443、4949、27000 ~ 27018)。

一旦完成了所有这些步骤，则可以单击 Verify Agent 按钮。如果所有一切都如预期般运行正常，则可以提交验证，将看到一个 Continue 按钮。

当单击 Continue 按钮时，将看到图 5-7 中所示的 Review and Deploy 页面，可以在其中看到准备被部署的所有处理过程。此处一个自动化操作代理会下载并安装监控和备份代理。

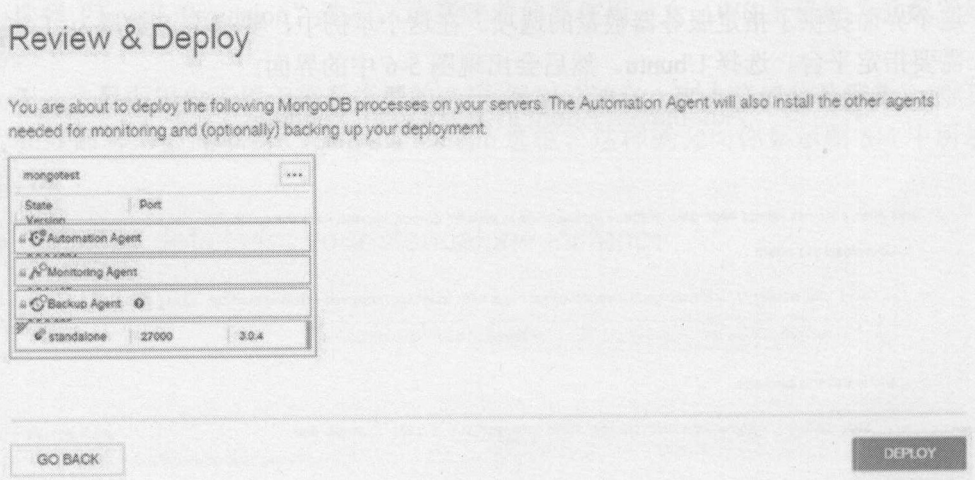

图 5-7 检查和部署

单击 Deploy 按钮会显示部署页面，其中部署状态会变更为“In progress”。当安装完成时，该部署状态将变更为“Goal State”，并且配置好的服务器将出现在拓扑视图中。

如果你的部署支持 SSL 或者使用任何身份验证机制，就需要手动下载并安装一个监控代理。

为了审查是否所有的代理都在正常工作，可以在控制台上单击 Administration 选项卡。

云管理器可以部署 MongoDB 副本集、分片群集，以及位于任意连接互联网的服务器上的独立部署。这些服务器只要能够让出站 TCP 连接到云管理器即可。

5.9 本章小结

在本章中，你学习了如何在 Windows 和 Linux 平台上安装 MongoDB。你还了解了确保数据库安全性以及安全使用的必要的一些重要配置。本章结尾处介绍了使用 MongoDB 云管理器进行配置的内容。

在下一章中，将开始使用 MongoDB Shell。

第 6 章

使用 MongoDB Shell

“MongoDB 的标准发行版本中提供了 mongo shell。它提供了一种具有对 JavaScript 语言和标准函数的完全访问权限的 JavaScript 环境。它为 MongoDB 数据库提供了一个完整接口。”

在本章中，将学习 mongo shell 的基础知识以及如何使用它来管理 MongoDB 文档。在你深入探究创建与数据库交互的应用程序之前，理解 MongoDB shell 如何工作是很重要的。

没有比上手使用 MongoDB shell 更好的体验 MongoDB 数据库的方法了。我们将 MongoDB shell 介绍分成三个部分，以便读者能够更加容易地领会和实践这些概念。

第一节涵盖了数据库的基础功能，其中包括基本的 CRUD 操作符。接下来的一节将介绍高级的查询。本章的最后一节会阐释存储和检索数据的两种方式：嵌入和引用。

6.1 基本查询

本节将简要探讨 CRUD 操作符(创建、读取、更新和删除)。使用基础示例和练习，将学习这些操作在 MongoDB 中是如何执行的。另外，将理解查询是如何在 MongoDB 中执行的。

与用于查询的传统 SQL 相反，MongoDB 使用了其自己的类似于 JSON 的查询语言来从存储数据中检索信息。

正如第 5 章中所阐释的，在成功安装 MongoDB 之后，将导航到目录 C:\practicalmongodb\bin\。这个文件夹具有用于运行 MongoDB 的所有可执行程序。

可以通过执行 mongo 可执行程序来启动 MongoDB shell。

第一步总是要启动数据库服务器。打开命令行提示符(以管理员身份运行它)并且运行命令 CD \。

接下来，运行命令 C:\practicalmongodb\bin\mongod.exe。(如果该安装位于其他某个文件夹，那么这个路径将相应变更。对于本章中的示例来说，安装位于 C:\practicalmongodb 文件夹)。这样就会启动数据库服务器。

```
C:\>c:\practicalmongodb\bin\mongod.exe
```

```
    2015-07-06T02:29:24.501-0700 I CONTROL Hotfix KB2731284 or later update
is insalled, no need to zero-out data files
    2015-07-06T02:29:24.522-0700 I JOURNAL [initandlisten] journal dir=c:\
data\db\ournal
    ..................................................
    2015-07-06T02:29:24.575-0700 I CONTROL [initandlisten] MongoDB starting :
pid=384 port=27017 dbpath=c:\data\db\ 64-bit host=ANC09
    2015-07-06T02:29:24.575-0700 I CONTROL [initandlisten] targetMinOS:
Windows 7/windows Server 2008 R2
    2015-07-06T02:29:24.575-0700 I CONTROL [initandlisten] db version v3.0.4
    2015-07-06T02:29:24.575-0700 I CONTROL [initandlisten] OpenSSL version:
OpenSSL1.0.1j-fips 19 Mar 2015
    2015-07-06T02:29:24.575-0700 I CONTROL [initandlisten] build info:
windows sys getwindowsversion(major=6, minor=1, build=7601, platform=2,
service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
    2015-07-06T02:29:24.575-0700 I CONTROL [initandlisten] allocator: system
    2015-07-06T02:29:24.575-0700 I CONTROL [initandlisten] options: {}
    2015-07-06T02:29:24.584-0700 I NETWORK [initandlisten] waiting for
connections on port 27017
```

MongoDB 会默认侦听本地主机接口的 27017 端口上所有的入站连接。

既然数据库服务器已经启动了，那么你就可以开始使用 mongo shell 将命令发送到该服务器。

在你查看 mongo shell 之前，我们简要了解如何使用导入/导出工具来将数据导入和导出 MongoDB 数据库。

首先，创建一个 CSV 文件来保存具有以下结构的学生记录：

```
Name, Gender, Class, Score, Age.
```

图 6-1 中显示了该 CSV 的样本数据。

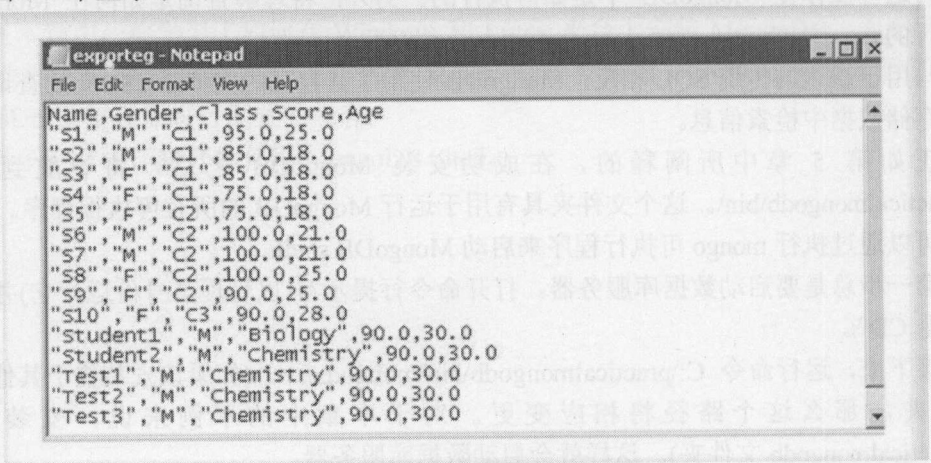

图 6-1 CSV 样本文件

接下来，将数据从 MongoDB 数据库导入到一个新的集合，以便了解该导入工具如何工作。

以管理员身份打开命令行提示符来运行它。以下命令被用于获得关于导入命令的帮助：

```
C:\>c:\practicalmongodb\bin\mongoimport.exe --help
Import CSV, TSV or JSON data into MongoDB.
When importing JSON documents, each document must be a separate line of
the input file.
Example:
  mongoimport --host myhost --db my_cms --collection docs < mydocfile.json
....
C:\>
```

运行以下命令来将数据从 exporteg.csv 文件导入到 MyDB 数据库中一个新的名称为 importeg 的集合中：

```
C:\>c:\practicalmongodb\bin\mongoimport.exe --host localhost --db mydb
--collection
importeg --type csv --file c:\exporteg.csv --headerline
2015-07-06T01:53:23.537-0700    connected to: localhost
2015-07-06T01:53:23.608-0700    imported 15 documents
```

为了验证该集合是否被创建以及数据是否被导入，要使用 mongo shell 连接到该数据库(在本示例中是本地主机)，并且你要运行命令来验证该集合是否存在。

要启动 mongo shell，需要以管理员身份运行命令行提示符并且输入命令 C:\PracticalMongoDB\bin\mongo.exe(该路径将因安装文件夹而异；在这个示例中，该文件夹是 C:\PracticalMongoDB\)，然后按下 Enter 键。

默认情况下，这会连接到在端口 27017 上进行侦听的 localhost 数据库服务器。

```
C:\>c:\practicalmongodb\bin\mongo.exe
MongoDB shell version: 3.0.4
connecting to: test
> use mydb
switched to db mydb
> show collections
importeg
system.indexes
>db.importeg.find()
{ "_id" : ObjectId("5450af58c770b7161eefd31d"), "Name" : "S1", "Gender" : "M",
"Class" : "C1", "Score" : 95, "Age" : 25 }
.......

{ "_id" : ObjectId("5450af59c770b7161eefd31e"), "Name" : "S2", "Gender" : "M",
"Class" : "C1", "Score" : 85, "Age" : 18 }
>
```

简要来说，此处你正在做的就是：

(1) 连接到 mongo shell。

(2) 切换到你的数据库，本示例中是 MyDB。

(3) 使用 show collections 检查存在于 MyDB 数据库中的集合。

(4) 使用导入工具检查你导入的集合数量。

(5) 最后，执行 find()命令来检查新集合中的数据。

要连接到不同的主机和端口，可以将–host 和–port 与命令一起使用。

正如可以在图 6-1 中所看到的，默认情况下数据库 test 被用于上下文环境。

在任何时候，执行 db 命令都会显示当前 shell 连接的数据库：

```
>db
test
>
```

为了显示所有的数据库名称，可以运行 show dbs 命令。执行这个命令将列出用于所连接服务器的所有数据库。

```
> show dbs
```

在任何时候，都可以使用 help()命令来获得帮助。

```
> help
        db.help()                 help on db methods
        db.mycoll.help()          help on collection methods
        sh.help()                 sharding helpers
        rs.help()                 replica set helpers
        help admin                administrative help
        help connect              connecting to a db help
        help keys                 key shortcuts
        help misc                 misc things to know
        help mr                   mapreduce
        show dbs                  show database names
        show collections          show collections in current database
        show users                show users in current database
.............
        exit                      quit the mongo shell
```

如上所示，如果需要关于 db 或 collection 的任何方法的帮助，则可以使用 db.help()或 db.<CollectionName>.help()。例如，如果需要关于 db 命令的帮助，则可以执行 db.help()。

```
>db.help()
DB methods:
        db.addUser(userDocument)
...
        db.shutdownServer()
        db.stats()
```

```
        db.version() current version of the server
>
```

到目前为止，你一直在使用默认的 test db。可以使用 use <newdbname>命令来切换到一个新的数据库。

```
> use mydb
switched to db mydb
```

在开始探究之前，我们首先简要了解与 SQL 专业术语和概念对应的 MongoDB 专业术语和概念。表 6-1 中汇总了这些内容。

表 6-1 SQL 和 MongoDB 专业术语

SQL	MongoDB
数据库	数据库
表	集合
行	文档
列	字段
索引	索引
表内联结	嵌入和引用
主键：可以指定一列或列的集合	主键：自动设置到_id 字段

我们现在开始探究 MongoDB 中用于查询的选项。切换到 MYDBPOC 数据库。

```
> use mydbpoc
switched to db mydbpoc
>
```

这样就会将上下文从 test 切换到 MYDBPOC。使用 db 命令可以确认这一点。

```
>db
mydbpoc
>
```

尽管上下文被切换到了 MYDBPOC，但如果运行 show dbs 命令，也不会显示该数据库名称，因为 MongoDB 只有在将数据插入到数据库时才会创建该数据库。这与 MongoDB 用于数据助益、动态命名空间分配以及简化和加速的开发过程的动态方式是一致的。如果此时运行 show dbs 命令，那么将不会在数据库列表中列出 MYDBPOC 数据库，因为该数据库只有在数据被插入时才会被创建。

以下示例假设了一个名称为 users 的多态集合，它包含以下两种原型文档：

```
{
id: ObjectID(),
```

```
FName: "First Name",
LName: "Last Name",
Age: 30, Gender: "M",
Country: "Country"
}
and
{
id: ObjectID(),
Name: "Full Name",
Age: 30,
Gender: "M",
Country: "Country"
}
and
{
id: ObjectID(), Name: "Full Name", Age: 30 }
```

6.1.1 创建和插入

你现在要查看数据库和集合是如何被创建的。正如较早所阐释的，MongoDB 中的文档都是 JSON 格式的。

首先，通过运行 db 命令将确认当前上下文是 mydbpoc 数据库。

```
>db
mydbpoc
>
```

现在将看到如何创建文档。

第一个文档遵循第一个原型，而第二个文档遵循第二个原型。你已经创建了两个名称为 user1 和 user2 的文档。

```
> user1 = {FName: "Test", LName: "User", Age:30, Gender: "M", Country: "US"}
{
        "FName" : "Test",
        "LName" : "User",
        "Age" : 30,
        "Gender" : "M",
        "Country" : "US"
}
> user2 = {Name: "Test User", Age:45, Gender: "F", Country: "US"}
{ "Name" : "Test User", "Age" : 45, "Gender" : "F", "Country" : "US" }
>
```

接着你要以下面的操作顺序将这两个文档(user1 和 user2)添加到 users 集合：

```
>db.users.insert(user1)
```

```
>db.users.insert(user2)
>
```

上述操作不仅会将这两个文档插入到 users 集合，还会创建该集合以及数据库。可以使用 show collections 和 show dbs 命令来验证这一点。

正如所提到过的，show dbs 将显示数据库列表。

```
> show dbs
admin     0.078GB
local     0.078GB
mydb      0.078GB
mydbproc  0.078GB
```

而 show collections 将显示当前数据库中的集合列表。

```
> show collections
system.indexes
users
>
```

system.indexes 集合也会与 users 集合一起显示。system.indexes 集合会在数据库被创建时默认创建。它管理数据库中所有集合的所有索引的信息。

执行命令 db.users.find()将显示 users 集合中的文档。

```
>db.users.find()
{ "_id" : ObjectId("5450c048199484c9a4d26b0a"), "FName" : "Test", "LName" :
"User", "Age" : 30, "Gender": "M", "Country" : "US" }
{ "_id" : ObjectId("5450c05d199484c9a4d26b0b"), "Name" : "Test", User",
"Age" : 45, "Gender" : "F", "Country" : "US" }

>
```

可以看到所创建的两个文档都显示出来了。除了你添加到文档的字段以外，还会为所有文档生成一个附加的_id 字段。

所有的文档都必须具有一个唯一的_id 字段。如果没有显式指定，那么 MongoDB 同样会自动分配一个作为唯一对象 ID，就像上面的示例中所显示的那样。

你没有显式插入一个 id 字段，但当使用 find()命令来显示文档时，可以看到一个与每个文档相关联的_id 字段。

其背后的原因在于，默认情况下索引是被创建在该_id 字段上的，这可以通过在 system.indexes 集合上运行 find 命令来验证。

```
>db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "mydbpoc.users", "name" : "_id_" }
>
```

可以使用 ensureIndex()和 dropIndex()命令将索引添加到集合中或者从中移除。在本章后面的内容中我们将介绍这一点。默认情况下，索引会被创建在所有集合的_id 字段上。这一默认索引无法被删除。

6.1.2 显式创建集合

在上面的示例中，第一个插入操作隐式地创建了集合。不过，用户也可以在执行插入语句之前显式创建一个集合。

```
db.createCollection("users")
```

6.1.3 使用循环插入文档

也可以使用一个 for 循环来将文档添加到集合。以下代码使用了 for 来插入用户。

```
>for(var i=1; i<=20; i++) db.users.insert({"Name" : "Test User" + i, "Age":
10+i, "Gender" : "F", "Country" : "India"})
>
```

为了验证插入是否成功，可以在集合上运行 find 命令。

```
>db.users.find()
{ "_id" : ObjectId("52f48cf474f8fdcfcae84f79"), "FName" : "Test", "LName" :
"User", "Age" : 30, "Gender" : "M", "Country" : "US" }
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" : "Test User",
"Age" : 45, "Gender" : "F", "Country" : "US" }
.................
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8c"), "Name" : "Test User18",
"Age" : 28, "Gender" : "F", "Country" : "India" }
Type "it" for more
>
```

用户出现在集合中。在你继续学习之前，我们要理解“Type “it” for more”语句。

find 命令会返回指向结果集的一个游标。相较于一次性在界面上显示所有文档(可能会有数千或数百万个结果)，游标会显示前 20 个文档并且等待请求遍历(it)以便显示其后 20 个文档，以此类推，直到所有的结果集都被显示出来。

所产生的游标还可以被分配到一个变量，然后可以使用一个 while 循环编程式地遍历它。该游标对象还可以作为一个数组来操作。

在你的例子中，如果输入“it”并且按下 Enter 键，就会出现以下信息：

```
>it
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8d"), "Name" : "Test User19",
"Age" : 29, "Gender" : "F", "Country" : "India" }
{ "_id" : ObjectId("52f48eeb74f8fdcfcae84f8e"), "Name" : "Test User20",
```

```
"Age" : 30, "Gender" : "F", "Country" : "India" }
    >
```

由于只有两个文档剩下，因此它会显示这剩下的两个文档。

6.1.4 通过显式指定_id 进行插入

在前面关于插入的示例中，并没有指定_id 字段，因此它是被隐式添加的。在以下示例中，将看到如何在一个集合中插入文档时显式指定_id 字段。

在显式指定_id 字段时，你必须注意该字段的唯一性；否则插入将会失败。

以下命令会显式指定_id 字段：

```
>db.users.insert({"_id":10, "Name": "explicit id"})
```

该插入操作会在 users 集合中创建以下文档：

```
{ "_id" : 10, "Name" : "explicit id" }
```

这一点可以通过运行以下命令来确认：

```
>db.users.find()
```

6.1.5 更新

在本节中将探究 update()命令，它被用于更新一个集合中的文档。

update()方法默认会更新单个文档。如果需要更新所有符合选择条件的文档，那么可以通过设置 multi 选项为 true 来实现这一目的。

我们首先更新已有列的值。$set 操作符将被用于更新记录。

以下命令会将所有女性用户的国家更新为 UK：

```
>db.users.update({"Gender":"F"}, {$set:{"Country":"UK"}})
```

要检查该更新是否完成，可以运行一个 find 命令来检查所有的女性用户。

```
    >db.users.find({"Gender":"F"})
    { "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" : "Test User",
"Age" : 45, "Gender" : "F", "Country" : "UK" }
    { "_id" : ObjectId("52f48eeb74f8fdcfcae84f7b"), "Name" : "Test User1",
"Age" : 11, "Gender" : "F", "Country" : "India" }
    { "_id" : ObjectId("52f48eeb74f8fdcfcae84f7c"), "Name" : "Test User2",
"Age" : 12, "Gender" : "F", "Country" : "India" }
    ...................
    Type "it" for more
    >
```

如果检查该输出结果，就会发现只更新了第一个文档记录，这是更新的默认行为，

因为没有指定 multi 选项。

现在我们修改该 update 命令并且加入 multi 选项：

```
>db.users.update({"Gender":"F"},{$set:{"Country":"UK"}},{multi:true})
>
```

再次运行 find 命令以检查是否已经为所有的女性雇员更新了国家。运行 find 命令将返回以下输出结果：

```
>db.users.find({"Gender":"F"})
{ "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a"), "Name" : "Test User",
"Age" : 45, "Gender" : "F", "Country" : "UK" }
.............
Type "it" for more
>
```

如你所见，已经将所有符合条件记录的国家更新为 UK。

在处理一个真实的应用程序时，你可能会碰到模式演化的情况，其中你可能最终会将字段添加到文档或者从中移除字段。我们来看看如何在 MongoDB 数据库中执行这些修改。

update()操作可被用于文档级别，这有助于更新一个集合中的单个文档或者文档集。

接下来，我们来看看如何将新字段添加到文档。为了将字段添加到文档，需要使用带有$set 操作符和 multi 选项的 update()命令。

如果用$set 使用一个字段名称，而该字段不存在，那么这个字段将会被添加到文档。以下命令会将字段 company 添加到所有的文档：

```
>db.users.update({},{$set:{"Company":"TestComp"}},{multi:true})
>
```

针对用户的集合运行 find 命令，将看到这个新的字段被添加到了所有的文档。

```
>db.users.find()
{ "Age" : 30, "Company" : "TestComp", "Country" : "US", "FName" : "Test",
"Gender" : "M", "LName" : "User", "_id" : ObjectId("52f48cf474f8fdcfcae84f79") }
{ "Age" : 45, "Company" : "TestComp", "Country" : "UK", "Gender" : "F",
"Name" : "Test User", "_id" : ObjectId("52f48cfb74f8fdcfcae84f7a") }
{ "Age" : 11, "Company" : "TestComp", "Country" : "UK", "Gender" :
"F", ..................
Type "it" for more
>
```

如果对已经存在于文档中的字段执行 update()命令，那么将会更新该字段的值；不过，如果该字段并不存在于文档中，那么该字段将会被添加到文档。

接下来将看到如何使用带有$unset 操作符的相同 update()命令来从文档中移除字段。

以下命令会将字段 Company 从所有的文档中移除：

```
>db.users.update({},{$unset:{"Company":""}},{multi:true})
>
```

这可以通过针对 Users 集合运行 find()命令来检查。可以看到，Company 字段已经从文档中删除了。

```
>db.users.find()
{ "Age" : 30, "Country" : "US", "FName" : "Test", "Gender" : "M", "LName" :
"User", "_id" :
ObjectId("52f48cf474f8fdcfcae84f79") }
.............
Type "it" for more
```

6.1.6 删除

要删除一个集合中的文档，可以使用 remove()方法。如果指定一个选择条件，那么只有满足该条件的文档才会被删除。如果没有指定条件，则会删除所有的文档。

以下命令将会删除 Gender = 'M'的文档：

```
>db.users.remove({"Gender":"M"})
>
```

可以通过在 Users 上运行 find()命令来验证这一点：

```
>db.users.find({"Gender":"M"})
>
```

不会返回任何文档。

以下命令将删除所有的文档：

```
>db.users.remove({})
>db.users.find()
>
```

如你所见，不会返回任何文档。

最后，如果希望删除该集合，那么以下命令将会删除这个集合：

```
>db.users.drop()
true
>
```

为了验证该集合是否被删除，可以运行 show collections 命令。

```
> show collections
system.indexes
>
```

如你所见，没有显示集合名称，这样就可以确认该集合已经从数据库中移除了。

我们已经介绍了基本的创建、更新和删除操作，下一节将介绍如何执行读取操作。

6.1.7 读取

在本章的这一部分将了解各种示例，这些示例揭示了作为 MongoDB 的一部分当前可用的查询功能，这些功能使得你能够从数据库中读取所存储的数据。

为了开始使用基本查询，首先创建 users 集合并且使用 insert 命令插入数据。

```
> user1 = {FName: "Test", LName: "User", Age:30, Gender: "M", Country: "US"}
{
        "FName" : "Test",
        "LName" : "User",
        "Age" : 30,
        "Gender" : "M",
        "Country" : "US"
}
> user2 = {Name: "Test User", Age:45, Gender: "F", Country: "US"}
{ "Name" : "Test User", "Age" : 45, "Gender" : "F", "Country" : "US" }
>db.users.insert(user1)
>db.users.insert(user2)
>for(var i=1; i<=20; i++) db.users.insert({"Name" : "Test User" + i, "Age":
10+i, "Gender" : "F", "Country" : "India"})
```

现在开始使用基本查询。find()命令被用于从数据库中检索数据。

运行 find()命令会返回集合中的所有文档。

```
>db.users.find()
{ "_id" : ObjectId("52f4a823958073ea07e15070"), "FName" : "Test", "LName" :
"User", "Age" : 30, "Gender" : "M", "Country" : "US" }
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User",
"Age" : 45, "Gender" : "F", "Country" : "US" }
......
{ "_id" : ObjectId("52f4a83f958073ea07e15083"), "Name" : "Test User18",
"Age" :28, "Gender" : "F", "Country" : "India" }
Type "it" for more
>
```

1. 查询文档

MongoDB 提供了一个富查询系统。查询文档可以作为参数被传递到 find()方法来过滤一个集合中的文档。

一个查询文档是在前“{”和后“}”一对大括号中指定的。一个查询文档是在返回结果集之前针对集合中的所有文档来匹配的。

使用不带有任何查询文档的 find()命令或者带有一个空查询文档的像 find({})这样的

命令会返回集合中所有的文档。

一个查询文档可以包含选择器和投影器。

选择器就像 SQL 中的 where 条件或者一个用于过滤出结果的过滤器。

投影器就像选择条件或者用于显示数据字段的选择列表。

2. 选择器

你现在将看到如何使用选择器。以下命令将返回所有的女性用户：

```
>db.users.find({"Gender":"F"})
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User",
"Age" : 45, "Gender" : "F", "Country" : "US" }
.............
{ "_id" : ObjectId("52f4a83f958073ea07e15084"), "Name" : "Test User19",
"Age" :29, "Gender" : "F", "Country" : "India" }
Type "it" for more
>
```

我们更进一步地阐释它。MongoDB 还支持将不同条件合并到一起的操作符以便根据你的需求来改进你的搜索。

现在我们将上面的查询修改为查询来自印度的女性用户。以下命令将返回相同结果：

```
>db.users.find({"Gender":"F", $or: [{"Country":"India"}]})
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1",
"Age" : 11, "Gender" : "F", "Country" : "India" }
...........
{ "_id" : ObjectId("52f4a83f958073ea07e15085"), "Name" : "Test User20",
"Age" :30, "Gender" : "F", "Country" : "India" }
>
```

接下来，如果希望找出所有来自印度或美国的女性用户，则可以执行以下命令：

```
>db.users.find({"Gender":"F",$or:[{"Country":"India"},{"Country":"US"}]})
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User",
"Age" : 45, "Gender" : "F", "Country" : "US" }
.............
{ "_id" : ObjectId("52f4a83f958073ea07e15084"), "Name" : "Test User19",
"Age" :29, "Gender" : "F", "Country" : "India" }
Type "it" for more
```

出于聚合的需要，则需要使用聚合函数。接着，将学习如何将 count()函数用于聚合。

在上面的示例中，相较于显示文档，你希望计算出生活在印度或美国的女性用户数。因此要执行以下命令：

```
>db.users.find({"Gender":"F",$or:[{"Country":"India"}, {"Country":"US"}]})
.count()
21
```

```
>
```

如果希望计算用户的计数而不考虑任何选择器，那么可以执行以下命令：

```
>db.users.find().count()
22
>
```

3. 投影器

你已经看到了如何使用选择器来过滤出集合中的文档。在上面的示例中，find()命令会返回匹配该选择器的文档的所有字段。

我们来将一个投影器添加到该查询文档，其中，除了该选择器之外，你还将涉及需要被显示的具体细节或者字段。

假定你希望显示所有女性员工的名字和年龄。在这种情况下，就还需要将一个投影器与该选择器一起使用。

执行以下命令以返回期望的结果集：

```
>db.users.find({"Gender":"F"}, {"Name":1,"Age":1})
{ "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User",
"Age" : 45 }
.....
Type "it" for more
>
```

4. sort()

在 MongoDB 中，排列顺序是按如下来指定的：1 用于升序排列，而-1 用于降序排列。

如果在上面的示例中，你希望按照年龄升序排列记录，那么你就要执行以下命令：

```
>db.users.find({"Gender":"F"}, {"Name":1,"Age":1}).sort({"Age":1})
{ "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1",
"Age" : 11 }
{ "_id" : ObjectId("52f4a83f958073ea07e15073"), "Name" : "Test User2",
"Age" : 12 }
{ "_id" : ObjectId("52f4a83f958073ea07e15074"), "Name" : "Test User3",
"Age" : 13 }
.............
{ "_id" : ObjectId("52f4a83f958073ea07e15085"), "Name" : "Test User20",
"Age" :30 }
Type "it" for more
```

如果希望按照名字降序且按照年龄升序显示记录，那么你就要执行以下命令：

```
>db.users.find({"Gender":"F"},{"Name":1,"Age":1}).sort({"Name":-1,"Age
":1})
```

```
    { "_id" : ObjectId("52f4a83f958073ea07e1507a"), "Name" : "Test User9",
"Age" : 19 }
    ............
    { "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1",
"Age" : 11 }
    Type "it" for more
```

5. limit()

你现在将了解如何才能限制你的结果集中的记录。例如，在具有数千个文档的大型集合中，如果仅希望返回 5 个匹配的文档，则可以使用 limit 命令，它完全可以让你完成该任务。

回到之前对生活在印度或美国的女性用户的查询，假如你希望限制该结果集并且只返回两个用户，则需要执行以下命令：

```
    >db.users.find({"Gender":"F",$or:[{"Country":"India"},{"Country":"US"}]}).
limit(2)
    { "_id" : ObjectId("52f4a826958073ea07e15071"), "Name" : "Test User",
"Age" : 45, "Gender" : "F", "Country" : "US" }
    { "_id" : ObjectId("52f4a83f958073ea07e15072"), "Name" : "Test User1",
"Age" : 11, "Gender" : "F", "Country" : "India" }
```

6. skip()

如果需求是跳过前两个记录并且返回第 3 和第 4 个用户，则可以使用 skip 命令。需要执行以下命令：

```
    >db.users.find({"Gender":"F",$or:[{"Country":"India"}, {"Country":"US"}]}).
limit(2).skip(2)
    { "_id" : ObjectId("52f4a83f958073ea07e15073"), "Name" : "Test User2",
"Age" : 12, "Gender" : "F", "Country" : "India" }
    { "_id" : ObjectId("52f4a83f958073ea07e15074"), "Name" : "Test User3",
"Age" : 13, "Gender" : "F", "Country" : "India" }
    >
```

7. findOne()

findOne()命令类似于 find()命令。findOne()方法可以使用与 find()一样的参数，但不同于返回一个游标，它会返回单个文档。假设你希望返回一个要么生活在印度要么生活在美国的女性用户，可以使用以下命令来实现此目标：

```
    >db.users.findOne({"Gender":"F"}, {"Name":1,"Age":1})
    {
            "_id" : ObjectId("52f4a826958073ea07e15071"),
            "Name" : "Test User",
            "Age" : 45
```

```
}
>
```

同样，如果希望返回第一个记录而不考虑那个例子中的任何选择器的话，则可以使用 findOne()，它将返回集合中的第一个文档。

```
>db.users.findOne()
{
        "_id" : ObjectId("52f4a823958073ea07e15070"),
        "FName" : "Test",
        "LName" : "User",
        "Age" : 30,
        "Gender" : "M",
        "Country" : "US"}
```

8. 使用游标

在使用了 find()方法时，MongoDB 会将查询结果作为一个游标对象来返回。为了显示该结果，mongo shell 会遍历所返回的游标。

MongoDB 使得用户可以使用 find 方法的游标对象。在接下来的示例中，将看到如何将游标对象存储到一个变量中并且使用 while 循环来操作它。

假设你希望返回在美国的所有用户。为了实现该目标，你创建了一个变量，将 find() 的输出结果指派给了这个变量，该结果是一个游标，然后使用 while 循环遍历和打印该输出结果。

这段代码看起来应该如下所示：

```
>var c = db.users.find({"Country":"US"})
>while(c.hasNext()) printjson(c.next())
{
        "_id" : ObjectId("52f4a823958073ea07e15070"),
        "FName" : "Test",
        "LName" : "User",
        "Age" : 30,
        "Gender" : "M",
        "Country" : "US"
}
{
        "_id" : ObjectId("52f4a826958073ea07e15071"),
        "Name" : "Test User",
        "Age" : 45,
        "Gender" : "F",
        "Country" : "US"
}
>
```

next()函数会返回下一个文档。hasNext()函数会在一个文档存在时返回 true，而

printjson()会以 JSON 格式呈现该输出结果。

被分配给游标对象的变量还可以作为数组来操作。如果，相较于循环遍历该变量，你希望显示位于数组索引 1 的文档，则可以运行以下命令：

```
>var c = db.users.find({"Country":"US"})
>printjson(c[1])
{
      "_id" : ObjectId("52f4a826958073ea07e15071"),
      "Name" : "Test User",
.... "Gender" : "F",
     "Country" : "US"}
>
```

9. explain()

explain()函数可用于查看在执行一个查询时 MongoDB 数据库当前运行的步骤。从版本 3.0 开始，该函数的输出格式以及传递给该函数的参数已经发生了变化。它使用了一个可选的被称为 verbose 的参数，它会判定 explain 输出看起来应该是什么样。这些是详细级别的模式：allPlansExecution、executionStats 以及 queryPlanner。默认的详细级别模式是 queryPlanner，这意味着如果不做指定，则将默认为 queryPlanner。

以下代码涵盖了过滤 username 字段时执行的步骤：

```
>db.users.find({"Name":"Test User"}).explain("allPlansExecution")

"queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "mydbproc.users",
        "indexFilterSet" : false,
        "parsedQuery" : {
                "$and" : [ ]
        },
        "winningPlan" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "$and" : [ ]
                },
                "direction" : "forward"
        },
        "rejectedPlans" : [ ]
},
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 20,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 0,
```

```
        "totalDocsExamined" : 20,
        "executionStages" : {
                "stage" : "COLLSCAN",
                "filter" : {
                        "$and" : [ ]
                },
                "nReturned" : 20,
                "executionTimeMillisEstimate" : 0,
                "works" : 22,
                "advanced" : 20,
                "needTime" : 1,
                "needFetch" : 0,
                "saveState" : 0,
                "restoreState" : 0,
                "isEOF" : 1,
                "invalidates" : 0,
                "direction" : "forward",
                "docsExamined" : 20
        },
        "allPlansExecution" : [ ]
},
"serverInfo" : {
        "host" : " ANOC9",
        "port" : 27017,
        "version" : "3.0.4",
        "gitVersion" : "534b5a3f9d10f00cd27737fbcd951032248b5952"
},
"ok" : 1
```

如你所见，explain()输出结果会返回关于 queryPlanner、executionStats 和 serverInfo 的信息。正如上面代码显示的，该输出所返回的信息取决于所选择详细程度的模式。

你已经看到了如何执行基本查询、排序、限制等。你还看到了如何使用 while 循环操作结果集或者将结果集作为数组来操作。在下一节中，将了解索引以及如何才能在你的查询中使用它们。

6.1.8 使用索引

索引被用于为频繁使用的查询提供高性能读取操作。默认情况下，当一个集合被创建并且文档被添加到其中时，就会在该文档的_id 字段上创建一个索引。

在本节中，将了解如何创建不同类型的索引。我们首先使用 for 循环在一个名称为 testindx 的新集合中插入 1 百万个文档。

```
>for(i=0;i<1000000;i++){db.testindx.insert({"Name":"user"+i,"Age":Math
.floor(Math.random()*120)})}
```

接着，运行 find()命令抓取值为 user101 的一个 Name。运行 explain()命令检查 MongoDB 正在执行哪些步骤以便返回结果集。

```
>db.testindx.find({"Name":"user101"}).explain("allPlansExecution")

{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "mydbproc.testindx",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "Name" : {
                                "$eq" : "user101"
                        }
                },
                "winningPlan" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "Name" : {
                                        "$eq" : "user101"
                                }
                        },
                        "direction" : "forward"
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 645,
                "totalKeysExamined" : 0,
                "totalDocsExamined" : 1000000,
                "executionStages" : {
                        "stage" : "COLLSCAN",
                        "filter" : {
                                "Name" : {
                                        "$eq" : "user101"
                                }
                        },
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 20,
                        "works" : 1000002,
                        "advanced" : 1,
                        "needTime" : 1000000,
                        "needFetch" : 0,
                        "saveState" : 7812,
```

```
                "restoreState" : 7812,
                "isEOF" : 1,
                "invalidates" : 0,
                "direction" : "forward",
                "docsExamined" : 1000000
        },
        "allPlansExecution" : [ ]
},
"serverInfo" : {
        "host" : " ANOC9",
        "port" : 27017,
        "version" : "3.0.4",
        "gitVersion" : "534b5a3f9d10f00cd27737fbcd951032248b5952"
},
"ok" : 1
```

如你所见，数据库扫描了整个表。这会造成严重的性能影响并且是由于没有使用索引造成的。

1. 单键索引

我们在该文档的 Name 字段上创建一个索引。使用 ensureIndex()来创建这个索引。

```
>db.testindx.ensureIndex({"Name":1})
```

索引的创建需要几分钟的时间，这取决于服务器以及集合的大小。

我们来运行与你之前使用 explain()运行的相同的查询，以检查在索引创建之后数据库会执行哪些步骤。检查输出结果中的 n、nscanned 和 millis 字段。

```
>db.testindx.find({"Name":"user101"}).explain("allPathsExecution")

{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "mydbproc.testindx",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "Name" : {
                                "$eq" : "user101"
                        }
                },
                "winningPlan" : {
                        "stage" : "FETCH",
                        "inputStage" : {
                                "stage" : "IXSCAN",
                                "keyPattern" : {
                                        "Name" : 1
```

```
                        },
                        "indexName" : "Name_1",
                        "isMultiKey" : false,
                        "direction" : "forward",
                        "indexBounds" : {
                                "Name" : [
                                        "[\"user101\", \"user101\"]"
                                ]
                        }
                }
        },
        "rejectedPlans" : [ ]
},
"executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
        "executionStages" : {
                "stage" : "FETCH",
                "nReturned" : 1,
                "executionTimeMillisEstimate" : 0,
                "works" : 2,
                "advanced" : 1,
                "needTime" : 0,
                "needFetch" : 0,
                "saveState" : 0,
                "restoreState" : 0,
                "isEOF" : 1,
                "invalidates" : 0,
                "docsExamined" : 1,
                "alreadyHasObj" : 0,
                "inputStage" : {
                        "stage" : "IXSCAN",
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 0,
                        "works" : 2,
                        "advanced" : 1,
                        "needTime" : 0,
                        "needFetch" : 0,
                        "saveState" : 0,
                        "restoreState" : 0,
                        "isEOF" : 1,
                        "invalidates" : 0,
                        "keyPattern" : {
```

```
                                        "Name" : 1
                                },
                                "indexName" : "Name_1",
                                "isMultiKey" : false,
                                "direction" : "forward",
                                "indexBounds" : {
                                        "Name" : [
                                                "[\"user101\", \"user101\"]"
                                        ]
                                },
                                "keysExamined" : 1,
                                "dupsTested" : 0,
                                "dupsDropped" : 0,
                                "seenInvalidated" : 0,
                                "matchTested" : 0
                        }
                },
                "allPlansExecution" : [ ]
        },
        "serverInfo" : {
                "host" : "ANOC9",
                "port" : 27017,
                "version" : "3.0.4",
                "gitVersion" : "534b5a3f9d10f00cd27737fbcd951032248b5952"
        },
        "ok" : 1
}
>
```

正如可以在该结果中所看到的，没有进行表扫描。索引的创建会在查询执行时间方面产生显著的差异。

2. 复合索引

在创建一个索引时，你应该牢记，索引要覆盖你大多数的查询。如果有时仅查询 Name 字段并且有时同时查询 Name 和 Age 字段，那么在 Name 和 Age 字段上创建一个复合索引将会比在单一字段上创建索引更有益处，因为复合索引将覆盖这两种查询。

以下命令会在 testindx 集合的 Name 和 Age 字段上创建一个复合索引。

```
>db.testindx.ensureIndex({"Name":1, "Age": 1})
```

复合索引有助于 MongoDB 更有效地执行带有多个子句的查询。在创建一个复合索引时，还有一点非常重要，这就是要牢记字段将被用于第一个出现的精确匹配(比如 Name : "S_1")，其后是要用在范围中的字段(比如 Age : {"$gt":20})。

因此，上面的索引对于以下查询将很有帮助：

```
>db.testindx.find({"Name": "user5","Age":{"$gt":25}}).explain
("allPlansExecution")

{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "mydbproc.testindx",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "$and" : [
                                {
                                        "Name" : {
                                                "$eq" : "user5"
                                        }
                                },
                                {
                                        "Age" : {
                                                "$gt" : 25
                                        }
                                }
                        ]
                },
                "winningPlan" : {
                        "stage" : "KEEP_MUTATIONS",
                        "inputStage" : {
                                "stage" : "FETCH",
                                "filter" : {
                                        "Age" : {
                                                "$gt" : 25
                                        }
                                },
                                ..........................
                                        "indexBounds" : {
                                                "Name" : [
                                                  "[\"user5\", \"user5\"
                },
                "rejectedPlans" : [
                        {
                                 "stage" : "FETCH",
.....................................................
                                "indexName" : "Name_1_Age_1",
                                "isMultiKey" : false,
                                "direction" : "forward",
.....................................................
        "executionStats" : {
                "executionSuccess" : true,
```

```
                "nReturned" : 1,
                "executionTimeMillis" : 0,
                "totalKeysExamined" : 1,
                "totalDocsExamined" : 1,
.....................................................
                        "inputStage" : {
                                "stage" : "FETCH",
                                "filter" : {
                                        "Age" : {
                                                "$gt" : 25
                                        }
                                },
                                "nReturned" : 1,
                                "executionTimeMillisEstimate" : 0,
                                "works" : 2,
                                "advanced" : 1,
                "allPlansExecution" : [
                        {
                                "nReturned" : 1,
                                "executionTimeMillisEstimate" : 0,
                                "totalKeysExamined" : 1,
                                "totalDocsExamined" : 1,
                                "executionStages" : {
.....................................................
        "serverInfo" : {
                "host" : " ANOC9",
                "port" : 27017,
                "version" : "3.0.4",
                "gitVersion" : "534b5a3f9d10f00cd27737fbcd951032248b5952"
        },
        "ok" : 1
}
>
```

3. 对 sort 操作的支持

在 MongoDB 中，使用一个索引字段来对文档排序的 sort 操作会提供最佳的性能。

就像在其他数据库中一样，由于这一点，MongoDB 中的索引具有一个顺序。如果一个索引被用于访问文档，那么它将按照与索引相同的顺序来返回结果。

复合索引需要在对多个字段进行排序时创建。在一个复合索引中，输出结果可以按照索引前缀的顺序或者完整索引的顺序来排序。

索引前缀是复合索引的一个子集，它包含索引开头部分的一个或多个字段。

例如，{ x:1, y: 1, z: 1}就是复合索引的索引前缀。

sort 操作可以基于索引前缀的任意组合，比如{x: 1}, {x: 1, y: 1}。

如果一个复合索引是排序的一个前缀，那么它就仅能有助于排序。

例如，一个基于 Age、Name 和 Class 的复合索引，类似于

```
>db.testindx.ensureIndex({"Age": 1, "Name": 1, "Class": 1})
```

那么，它对于以下查询就是有用的：

```
>db.testindx.find().sort({"Age":1})
>db.testindx.find().sort({"Age":1,"Name":1})
>db.testindx.find().sort({"Age":1,"Name":1, "Class":1})
```

上面的索引在以下查询中没什么用处：

```
>db.testindx.find().sort({"Gender":1, "Age":1, "Name": 1})
```

可以通过使用 explain()命令来诊断 MongoDB 是如何处理一个查询的。

4. 唯一索引

在一个字段上创建索引并不会确保唯一性，因此，如果一个索引是基于 Name 字段创建的，那么两个或多个文档就可以具有相同的名称。不过，如果唯一性是需要被启用的其中一个约束，那么在创建该索引时，这个唯一属性就需要被设置为 true。

首先，我们删除已有的索引。

```
>db.testindx.dropIndexes()
```

以下命令将在 testindx 集合的 Name 字段上创建一个唯一索引：

```
>db.testindx.ensureIndex({"Name":1},{"unique":true})
```

现在，如果像下面所示的那样尝试在该集合中插入重复的名称，那么 MongoDB 就会返回一个错误，并且不允许重复记录的插入：

```
>db.testindx.insert({"Name":"uniquename"})
>db.testindx.insert({"Name":"uniquename"})
"E11000 duplicate key error index: mydbpoc.testindx.$Name_1 dup key: { :
"uniquename" }"
```

如果检查该集合，就会看到只保存了第一个 uniquename。

```
>db.testindx.find({"Name":"uniquename"})
{ "_id" : ObjectId("52f4b3c3958073ea07f092ca"), "Name" : "uniquename" }
>
```

也可以为复合索引启用唯一性，这意味着尽管单个字段可以具有重复值，但其组合将一直是唯一的。

例如，如果有一个{"name":1, "age":1}的索引，

```
>db.testindx.ensureIndex({"Name":1, "Age":1},{"unique":true})
```

```
>
```

那么以下插入将是允许的：

```
>db.testindx.insert({"Name":"usercit"})
>db.testindx.insert({"Name":"usercit", "Age":30})
```

不过，如果执行以下代码：

```
>db.testindx.insert({"Name":"usercit", "Age":30})
```

它将抛出一个错误，如以下代码所示：

```
E11000 duplicate key error index: mydbpoc.testindx.$Name_1_Age_1
dup key: { : "usercit", : 30.0 }
```

可以先创建集合并且插入文档，然后在该集合上创建一个索引。如果在该集合上创建一个唯一索引，而创建索引的字段可能存在重复值的话，那么索引的创建将会失败。

为了满足这一场景，MongoDB 提供了一个 dropDups 选项。这个 dropDups 选项会保留找到的第一个文档并且移除具有重复值的所有后续文档。

以下命令将在 name 字段上创建一个唯一索引，并且将删除所有重复的文档：

```
>db.testindx.ensureIndex({"Name":1},{"unique":true, "dropDups":true})
>
```

5. system.indexes

无论你何时创建一个数据库，默认情况下都会创建一个 system.indexes 集合。关于数据索引的所有信息都存储在 system.indexes 集合中。这是一个保留集合，因此你无法修改其文档或者从中移除文档。你只能通过 ensureIndex 和 dropIndexes 数据库命令来操作它。

无论何时创建了一个索引，都可以在 system.indexes 中看到其元信息。以下命令可用于提取所有关于示例集合的索引信息：

```
db.collectionName.getIndexes()
```

例如，以下命令将返回所有在 testindx 集合上创建的索引：

```
>db.testindx.getIndexes()
```

6. dropIndex

dropIndex 命令被用于移除索引。

以下命令将从 testindx 集合中移除 Name 字段索引：

```
>db.testindx.dropIndex({"Name":1})
{ "nIndexesWas" : 3, "ok" : 1 }
>
```

7. reIndex

当已经在集合上执行了若干插入和删除时，你可能必须重构索引，以便可以用最佳的方式使用索引。reIndex 命令被用于重构索引。

以下命令会重构一个集合的所有索引。它首先会删除索引，其中包括_id 字段上的默认索引，然后它会重构这些索引。

```
db.collectionname.reIndex()
```

以下命令会重构 testindx 集合的索引：

```
>db.testindx.reIndex()
{
        "nIndexesWas" : 2,
        "msg" : "indexes dropped for collection",
        "nIndexes" : 2,
        ..............
        "ok" : 1
}
>
```

我们将在下一章详尽探讨 MongoDB 中可用的不同类型的索引。

8. 索引如何工作

MongoDB 会在一个二叉树(BTree)结构中存储索引，因此它自动支持范围查询。

如果一个查询中使用了多个选择条件，那么 MongoDB 就会尝试找到最佳的单个索引来选择一个候选集。在那之后，它会依次遍历该集合来估算其他条件。

在首次执行该查询时，MongoDB 会为每一个可用于该查询的索引创建多个执行计划。它会让这些计划在一定的时间间隔中轮流执行，直到执行最快的计划完成。然后其结果会被返回到系统，该结果会记住最快执行计划使用的索引。

对于后续查询，将会使用此被记住的索引，直到该集合中发生了一定数量的更新。在超过更新限制时，系统将再次执行该处理过程以找出此时适用的最佳索引。

当发生以下任意事件时，就会再次对查询计划进行评估：

- 集合接收到 1000 个写操作。
- 添加或删除了一个索引。
- mongod 程序重启了。
- 发生了用于重构索引的再次索引。

如果希望重写 MongoDB 的默认索引选择，那么可以使用 hint()方法达成目的。

在版本 2.6 中引入了索引过滤器。它是由优化器将为一个查询所评估的索引组成的，评估对象包括查询、投影和排序。MongoDB 将使用由索引过滤器提供的索引并将忽略 hint()。

在版本 2.6 之前，MongoDB 一直都仅使用一个索引，因此你需要确保组合索引的存在，以更好匹配你的查询。这可以通过检查查询的排序以及搜索条件来完成。

索引交集是在版本 2.6 中引入的。它使得用于满足具有复合条件的查询的索引交集成为可能，其中一部分条件由一个索引满足，而其他部分则由其他索引来满足。

一般来说，索引交集是由两个索引组成的；不过，也可以将多个索引交集用于解决一个查询。此功能提供了更好的优化。

就像在其他数据库中一样，索引的维护总是伴随着附加成本。变更集合的每一个操作(比如创建、更新或删除)都会带来开销，因为索引也需要被更新。为了维持一个最佳平衡，你需要定期检查使用一个索引的有效性，这可以通过你在系统上执行的读和写的比例来衡量。识别出较少使用的索引并且删除它们。

6.2 进阶介绍

本节将介绍在选择器部分使用条件操作符和正则表达式的高级查询。这些操作符和正则表达式中的每一个都会提供对于你所编写的查询的更多控制权，并且相应的你也就拥有了对于能从 MongoDB 数据库中抓取到的信息的更多控制权。

6.2.1 使用条件操作符

条件操作符使你可以对尝试从数据库中提取的数据拥有更多的控制权。在本节中，将专注于以下操作符：$lt、$lte、$gt、$gte、$in、$nin 以及$not。

以下示例假设了一个名称为 Students 的集合，它包含以下文档类型：

```
{
_id: ObjectID(),
Name: "Full Name",
Age: 30,
Gender: "M",
Class: "C1",
Score: 95
}
```

首先要创建该集合并且插入一些样本文档。

```
>db.students.insert({Name:"S1",Age:25,Gender:"M",Class:"C1",Score:95})
>db.students.insert({Name:"S2",Age:18,Gender:"M",Class:"C1",Score:85})
>db.students.insert({Name:"S3",Age:18,Gender:"F",Class:"C1",Score:85})
>db.students.insert({Name:"S4",Age:18,Gender:"F",Class:"C1",Score:75})
>db.students.insert({Name:"S5",Age:18,Gender:"F",Class:"C2",Score:75})
>db.students.insert({Name:"S6",Age:21,Gender:"M",Class:"C2",Score:100})
>db.students.insert({Name:"S7",Age:21,Gender:"M",Class:"C2",Score:100})
```

```
>db.students.insert({Name:"S8",Age:25,Gender:"F",Class:"C2",Score:100})
>db.students.insert({Name:"S9",Age:25,Gender:"F",Class:"C2",Score:90})
>db.students.insert({Name:"S10",Age:28,Gender:"F",Class:"C3",Score:90})
>db.students.find()
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25,
"Gender" : "M", "Class" : "C1", "Score" : 95 }
.....................
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" :
28, "Gender" : "F", "Class" : "C3", "Score" : 90 }
>
```

1. $lt 和$lte

我们从$lt 和$lte 操作符开始讲解。它们分别表示“小于”和“小于等于”。

如果希望找出小于 25 岁(Age < 25)的所有学生，就可以执行以下具有一个选择器的 find 命令：

```
>db.students.find({"Age":{"$lt":25}})
{ "_id" : ObjectId("52f8750ca13cd6a65998734e"), "Name" : "S2", "Age" : 18,
"Gender" : "M", "Class" : "C1", "Score" : 85 }
.............................
{ "_id" : ObjectId("52f87556a13cd6a659987353"), "Name" : "S7", "Age" : 21,
"Gender" : "M", "Class" : "C2", "Score" : 100 }
>
```

如果希望找出小于或等于 25 岁(Age≤25)的所有学生，则可以执行以下命令：

```
>db.students.find({"Age":{"$lte":25}})
{ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25,
"Gender" : "M", "Class" : "C1", "Score" : 95 }
.....................
{ "_id" : ObjectId("52f87578a13cd6a659987355"), "Name" : "S9", "Age" : 25,
"Gender" : "F", "Class" : "C2", "Score" : 90 }
>
```

2. $gt 和$gte

$gt 和$gte 操作符分别表示“大于”和“大于等于”。

我们来找出 Age > 25 的所有学生。这可以通过执行以下命令来完成：

```
>db.students.find({"Age":{"$gt":25}})
{ "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" :
28, "Gender" : "F", "Class" : "C3", "Score" : 90 }
>
```

如果修改上面的示例以返回 Age ≥ 25 的学生，那么其命令就如下所示：

```
>db.students.find({"Age":{"$gte":25}})
```

```
    { "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25,
"Gender" : "M", "Class" : "C1", "Score" : 95 }
    ..................................
    { "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" :
28, "Gender" : "F", "Class" : "C3", "Score" : 90 }
    >
```

3. $in 和$nin

我们找出属于 C1 或 C2 班的所有学生。其命令如下所示：

```
    >db.students.find({"Class":{"$in":["C1","C2"]}})
    { "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" : "S1", "Age" : 25,
"Gender" : "M", "Class" : "C1", "Score" : 95 }
    ..............................
    { "_id" : ObjectId("52f87578a13cd6a659987355"), "Name" : "S9", "Age" : 25,
"Gender" : "F", "Class" : "C2", "Score" : 90 }
    >
```

与此相反的信息可以通过使用$nin 来返回。

我们接下来找出那些不属于 C1 或 C2 班的学生。其命令如下所示：

```
    >db.students.find({"Class":{"$nin":["C1","C2"]}})
    { "_id" : ObjectId("52f8758da13cd6a659987356"), "Name" : "S10", "Age" :
28, "Gender" : "F", "Class" : "C3", "Score" : 90 }
    >
```

接下来看看如何才能组合上述所有操作符并且编写一个查询。假设你希望找出性别是“M”或者属于“C1”或“C2”班并且年龄大于或等于 25 岁的所有学生。这可以通过执行以下命令来完成：

```
    >db.students.find({$or:[{"Gender":"M","Class":{"$in":["C1","C2"]}}],
"Age":{"$gte":25}}){ "_id" : ObjectId("52f874faa13cd6a65998734d"), "Name" :
"S1", "Age" : 25, "Gender" : "M", "Class" : "C1", "Score" : 95 }
    >
```

6.2.2 正则表达式

在本节中，将了解如何使用正则表达式。正则表达式在你希望找出姓名以“A”开头的学生这样的场景中是很有用的。

为了理解这一点，我们再添加 3 个或 4 个不同姓名的学生。

```
    >db.students.insert({Name:"Student1", Age:30, Gender:"M", Class:
"Biology", Score:90})
    >db.students.insert({Name:"Student2", Age:30, Gender:"M", Class:
"Chemistry", Score:90})
```

```
    >db.students.insert({Name:"Test1", Age:30, Gender:"M", Class:
"Chemistry", Score:90})
    >db.students.insert({Name:"Test2", Age:30, Gender:"M", Class:
"Chemistry", Score:90})
    >db.students.insert({Name:"Test3", Age:30, Gender:"M", Class:
"Chemistry", Score:90})
    >
```

假设你希望找出姓名以“St”或“Te”开头并且其班级以“Che”开头的所有学生。使用正则表达式就能筛选出这些结果，如以下代码所示：

```
    >db.students.find({"Name":/(St|Te)*/i, "Class":/(Che)/i})
    { "_id" : ObjectId("52f89ecae451bb7a56e59086"), "Name" : "Student2",
"Age" : 30, "Gender" : "M", "Class" : "Chemistry", "Score" : 90 }
    ........................
    { "_id" : ObjectId("52f89f06e451bb7a56e59089"), "Name" : "Test3", "Age" : 30,
    "Gender" : "M", "Class" : "Chemistry", "Score" : 90 }
    >
```

为了理解正则表达式如何工作，我们使用查询"Name":/(St|Te)*/i 作为例子。

//i 表明该正则表达式是不区分大小写的。

(St|Te)*意味着 Name 字符串必须以“St”或“Te”开头。

结尾处的*意味着它将匹配其后的所有内容。

当将所有这一切组合使用时，你就是在对以“St”或“Te”开头的名称进行不区分大小写的匹配。在用于 Class 的正则表达式中，也运行了相同正则表达式。

接下来，我们让这个查询更复杂一些。我们加入上面介绍的操作符。

抓取姓名如 student1、student2 并且其性别为男性以及年龄大于等于 25 岁的学生。其命令如下：

```
    >db.students.find({"Name":/(student*)/i,"Age":{"$gte":25},"Gender":"M"})
    { "_id" : ObjectId("52f89eb1e451bb7a56e59085"), "Name" : "Student1",
"Age" : 30, "Gender" : "M", "Class" : "Biology", "Score" : 90 }
    { "_id" : ObjectId("52f89ecae451bb7a56e59086"), "Name" : "Student2",
"Age" : 30, "Gender" : "M", "Class" : "Chemistry", "Score" : 90 }
```

6.2.3 MapReduce

MapReduce 框架使得任务分配成为可能，在这个示例中，就是数据跨计算机群集的聚合以便缩短用来聚合数据集的时间。它由两部分构成：映射(Map)和缩小(Reduce)。

这里是一个更具体的描述：MapReduce 是一个框架，它被用于处理跨大量数据集的高可分配问题以及使用多个节点来运行的问题。如果所有这些节点都使用相同的硬件，那么这些节点就被统称为一个群集；否则，它就会被称为一个网格。这一处理过程可以作用于结构化数据(数据库中存储的数据)以及非结构化数据(文件系统中存储的数据)。

- **“Map”**：在这个步骤中，该节点会充当接收输入参数的主节点并且将大问题划分成多个小的子问题。然后这些子问题会被跨工作节点进行分发。工作节点可能会进一步将问题划分成子问题。这就产生了多层树结构。然后工作节点将处理其中的子问题并且将答案返回给主节点。
- **“Reduce”**：在这个步骤中，所有的子问题答案都被提供给了主节点，然后主节点会合并所有这些答案并且生成最终的输出结果，这就是你尝试解决的大问题的答案。

为了理解它如何工作，我们思考一个小例子，其中你要找出集合中男学生和女学生的人数。

这涉及以下步骤：首先你要创建 map 和 reduce 函数，然后你要调用 mapReduce 函数并且传递必要的参数。

我们首先定义 map 函数：

```
>var map = function(){emit(this.Gender,1);};
>
```

这一步用作输入文档并且基于 Gender 字段来发送类型为{"F", 1}或{"M", 1}的文档。

接着，你要创建 reduce 函数：

```
>var reduce = function(key, value){return Array.sum(value);};
>
```

这将基于 key 字段来分组 map 函数所发送的文档，在本示例中该 key 字段就是 Gender，并且将返回值的合计，在本示例中这个值是作为“1”来发送的。上面定义的 reduce 函数的输出结果就是关于性别的统计计数。

最后，要使用 mapReduce 函数将它们放在一起，如下所示：

```
>db.students.mapReduce(map, reduce, {out: "mapreducecount1"})
{
        "result" : "mapreducecount1",
        "timeMillis" : 29,
        "counts" : {
                "input" : 15,
                "emit" : 15,
                "reduce" : 2,
                "output" : 2
        },
        "ok" : 1,
}
>
```

这实际上是在应用该 map、reduce 函数，它们是在 students 集合上定义的。最终的结果会被存储到一个名称为 mapreducecount1 的新集合中。

为了验证它，可以在 mapreducecount1 集合上运行 find()命令，如下所示：

```
>db.mapreducecount1.find()
{ "_id" : "F", "value" : 6 }
{ "_id" : "M", "value" : 9 }
>
```

这里还有一个用来说明 MapReduce 工作原理的示例。我们使用 MapReduce 来找出按班级统计的平均分。正如你在上面的示例中所看到的，首先需要创建 map 函数，然后创建 reduce 函数，最后你要将它们合并起来以将输出结果存储到数据库的一个集合中。其代码片段如下所示：

```
>var map_1 = function(){emit(this.Class,this.Score);};
>var reduce_1 = function(key, value){return Array.avg(value);};
>db.students.mapReduce(map_1,reduce_1, {out:"MR_ClassAvg_1"})
{
        "result" : "MR_ClassAvg_1",
        "timeMillis" : 4,
        "counts" : {
                "input" : 15, "emit" : 15,
                "reduce" : 3 , "output" : 5
        },
        "ok" : 1,
}

> db.MR_ClassAvg_1.find()
{ "_id" : "Biology", "value" : 90 }
{ "_id" : "C1", "value" : 85 }
{ "_id" : "C2", "value" : 93 }
{ "_id" : "C3", "value" : 90 }
{ "_id" : "Chemistry", "value" : 90 }
>
```

第一步是定义 map 函数，它会循环遍历集合文档并且将输出结果返回为{"Class": Score}，例如{"C1":95}。第二步会对班级进行分组并且计算该班级的平均分。第三步会合并结果；它会定义 map、reduce 函数需要被应用到的集合并且最终它会定义在何处存储该输出结果，在这个示例中，输出结果会被存储到一个名称为 MR_ClassAvg_1 的新集合中。

在最后一步中，使用了 find 以便检查产生的输出结果。

6.2.4 aggregate()

上一节介绍了 MapReduce 函数。在本节中，将粗浅介绍 MongoDB 的聚合框架。

聚合框架让你可以在不使用 MapReduce 函数的情况下算出聚合值。就性能方面来说，聚合框架比 MapReduce 函数要快。你总是需要牢记的是，MapReduce 是为了批量方法使用的，而非用于实时分析。

接下来要使用 aggregate 函数来描述上述两个探讨过的输出结果。首先，该输出结果

在于计算出男学生和女学生的人数。这可以通过执行以下命令来实现：

```
>db.students.aggregate({$group:{_id:"$Gender", totalStudent: {$sum: 1}}})
{ "_id" : "F", "totalStudent" : 6 }
{ "_id" : "M", "totalStudent" : 9 }
>
```

同样，为了计算出按班级统计的平均分，可以执行以下命令：

```
>db.students.aggregate({$group:{_id:"$Class", AvgScore: {$avg: "$Score"}}})
{ "_id" : "Biology", "AvgScore" : 90 }
{ "_id" : "C3", "AvgScore" : 90 }
{ "_id" : "Chemistry", "AvgScore" : 90 }
{ "_id" : "C2", "AvgScore" : 93 }
{ "_id" : "C1", "AvgScore" : 85 }
>
```

6.3 设计应用程序的数据模型

在本节中，将介绍如何为一个应用程序设计数据模型。MongoDB 数据库提供了两个选项用于设计一个数据模型：用户可以将相关对象彼此内嵌，或者可以使用 ID 来彼此引用。在本节中，将探究这些选项。

为了理解这些选项，要设计一个博客应用并且揭示这两个选项的使用。

一个典型的博客应用由以下应用场景构成：

人们会就不同的主题发表博文。除了主题分类之外，也可以使用不同的标签。举个例子，如果类别是政治，且博文谈论的内容与一名政治家有关，那么该政治家的姓名就可以添加到该博文作为一个标签。这有助于用户快速找到与其兴趣相关的博文，还可以让他们将相关的博文链接到一起。

浏览博文的人可以对博文进行评论。

6.3.1 关系型数据模型与标准化

在开始探究 MongoDB 的方法之前，我们先看看在像 SQL 这样的关系型数据库中你会如何对此进行建模。

在关系型数据库中，数据建模通常是通过定义表并且逐步消除数据冗余以实现标准形式来发展而成的。

1. 什么是标准形式

在关系型数据库中，标准形式通常由依据应用程序需求创建表开始，然后逐步消除冗余以实现最高的标准形式，该标准形式也被称为第三范式或 3NF。为了更好地理解这

一点，我们将博客应用程序的数据放入一张表单中。该初始数据如图 6-2 所示。

作者	博文	类别	标签	评论	评论者

图 6-2 博客应用程序的初始数据

这些数据实际上是第一范式。将有大量的冗余，因为可能会有针对博文的多条评论并且会有多个标签被关联到博文。当然，冗余带来的问题在于，它伴随着不一致的可能性，其中相同数据的各种副本可能具有不同的值。为了消除这一冗余，你需要通过将它划分成多个表以进一步标准化数据。作为此步骤的一部分，必须指定一个唯一标识表中每一行的键列，以便可以在表之间创建链接。在使用 3NF 对上述场景进行建模时，其标准形式看起来就像图 6-3 中所示的 RDBMS 图表一样。

在这个例子中，你有了一个没有冗余的数据模型，允许你在更新它时无须担心会更新多个行。尤其是，你不再需要担心数据模型中的不一致性了。

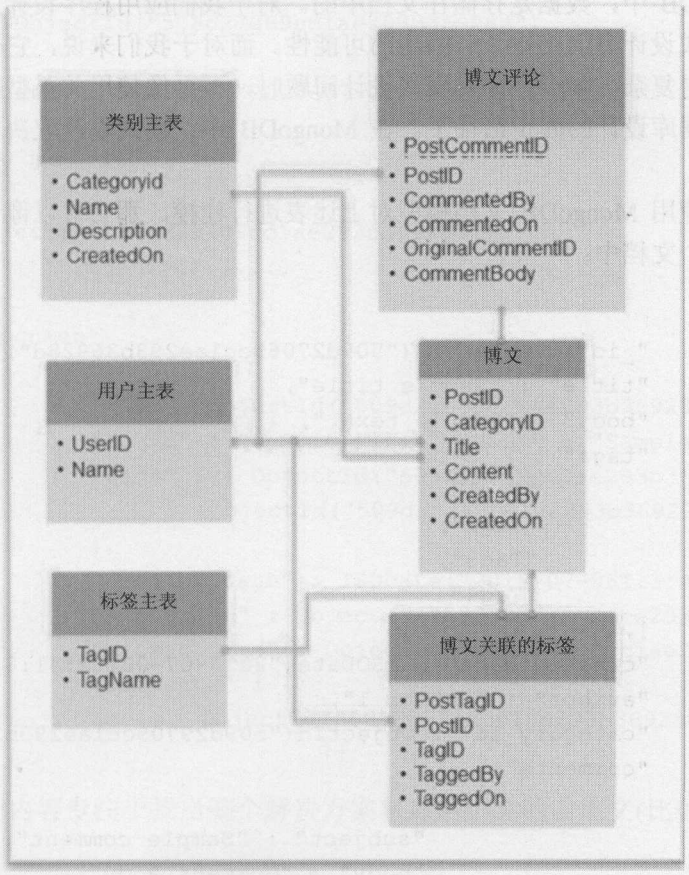

图 6-3 RDBMS 图表

2. 标准形式的问题

正如所提及的，标准化的好处在于，它允许在没有任何冗余的情况下轻易地进行更新(比如，它有助于保持数据一致性)。更新一个用户名称意味着更新 Users 表中的名称。

然而，当尝试取回数据时，会出现一个问题。例如，要找出与由特定用户发表的博文有关的所有标签和评论，关系型数据库编程人员就要使用 JOIN。通过使用 JOIN，数据库就会按照应用程序界面设计来返回所有的数据，但真正的问题在于数据库执行什么操作来得到该结果集。

通常，所有的 RDBMS 都会读取一个硬盘并且进行搜寻，这会将 99%的时间花费在读取一行上。在面临硬盘访问时，随机搜寻就是最大的敌人。在此背景下这一点如此重要的原因在于，JOIN 通常需要随机搜寻。JOIN 操作是关系型数据库中开销最大的操作之一。此外，如果最终需要将你的数据库扩展为多台服务器，那么你就会面临生成一个分布式联结的问题，这是一个复杂且通常很慢的操作。

6.3.2 MongoDB 文档数据模型方法

在 MongoDB 中，数据是存储在文档中的。对于我们应用程序设计人员来说，幸运的是，这在模式设计方面带来了一些新的可能性。而对于我们来说，它也会让我们的模式设计过程变得复杂。如今在面对模式设计问题时，不再像使用关系型数据库时那样有一种标准化数据库设计的固定路径了。在 MongoDB 中，模式设计取决于你试图解决的问题。

如果必须使用 MongoDB 文档模型对上述表进行建模，那么可以像下面这样将博文数据存储在一个文档中：

```
{
        "_id" : ObjectId("509d27069cc1ae293b36928d"),
        "title" : "Sample title",
        "body" : "Sample text.",
        "tags" : [
                "Tag1",
                "Tag2",
                "Tag3",
                "Tag4"
        ],
        "created_date" : ISODate("2015-07-06T12:41:39.110Z"),
        "author" : "Author 1",
        "category_id" : ObjectId("509d29709cc1ae293b369295"),
        "comments" : [
                {
                        "subject" : "Sample comment",
                        "body" : "Comment Body",
                        "author " : "author 2",
```

```
                    "created_date":ISODate("2015-07-06T13:34:
                23.929Z")
            }
        ]}
```

如你所见，将评论和标签仅仅嵌入到了单个文档中。或者，可以根据_id 字段通过引用评论和标签来稍微“标准化”一下这个模型：

```
// Authors document:
{
"_id": ObjectId("509d280e9cc1ae293b36928e "),
"name": "Author 1",}
// Tags document:
{
"_id": ObjectId("509d35349cc1ae293b369299"),
"TagName": "Tag1",.....}
// Comments document:
{
"_id": ObjectId("509d359a9cc1ae293b3692a0"),
"Author": ObjectId("508d27069cc1ae293b36928d"),
.......
"created_date" : ISODate("2015-07-06T13:34:59.336Z")
}
//Category Document
{
"_id": ObjectId("509d29709cc1ae293b369295"),
"Category": "Catgeory1"......
}
//Posts Document
        {
                "_id" : ObjectId("509d27069cc1ae293b36928d"),
                "title" : "Sample title","body" : "Sample text.",
                "tags" : [ ObjectId("509d35349cc1ae293b369299"),
                        ObjectId("509d35349cc1ae293b36929c")
                ],
                "created_date" : ISODate("2015-07-06T13:41:39.110Z"),
                "author_id" : ObjectId("509d280e9cc1ae293b36928e"),
                "category_id" : ObjectId("509d29709cc1ae293b369295"),
                "comments" : [
                        ObjectId("509d359a9cc1ae293b3692a0"),
                                ]}
```

本章剩下的内容专注于找出哪个解决方案将适用于你的上下文(比如是否使用引用或者是否嵌入)。

1. 嵌入

在本节中，将看到嵌入是否会对性能产生积极影响。当希望抓取一些数据集并且将它显示在界面上时，比如显示与博文有关的评论的页面，嵌入就会很有用；在这种情况下，评论可以被嵌入到 Blogs 文档中。

这种方法的好处在于，由于 MongoDB 会在硬盘上连续存储文档，因此可以在单次搜寻中抓取到所有相关的数据。

除此之外，由于不支持 JOIN 并且你在这个例子中使用引用，因此应用程序可以进行像下面这样的一些操作来抓取与博文相关的评论数据。

(1) 从 blogs 文档中抓取相关的评论_id。

(2) 根据第一步中找到的评论_id 抓取 comments 文档。

如果使用这种方法，也就是引用，那么不仅数据库必须进行多次搜寻来找到你的数据，而且会为查找带来额外的延迟，因为它现在会进行两次数据库访问以检索你的数据。

如果应用程序频繁访问与博文有关的评论数据，那么几乎可以肯定，在 blog 文档中嵌入评论将对性能产生积极影响。

衡量是否选用嵌入的另一个关注点是，在写入数据时对于原子性和隔离性的期望程度。MongoDB 的设计没有多文档事务。在 MongoDB 中，仅在单个文档级别提供了操作的原子性，因此需要以原子方式一起更新的数据需要被共同放置在单个文档中。

当更新数据库中的数据时，必须确保你的更新要么完全成功、要么完全失败，绝不能出现“部分成功”的情况，因而也就绝不会有其他数据库读取者看到未完成的写操作。

2. 引用

你已经看到了，嵌入是能够在许多情况下提供最佳性能的方法；它还会提供数据一致性的保障。然而，在有些情况下，一个更为标准的模型在 MongoDB 中会运行得更好。

使用多个集合并且添加引用的一个原因在于，在查询数据时它能提供增强的灵活性。我们用上面提到过的博客示例理解这一点。

你看到了如何使用嵌入模式，它对于在单个页面上同时显示所有数据来说将会很好地运行(比如显示带有所有相关评论的博文的页面)。

选择假定你需要搜索特定用户发表的评论。该查询(使用此嵌入模式)可能会如以下代码所示：

```
db.posts.find({'comments.author': 'author2'},{'comments': 1})
```

然后，这个查询的结果就会是下面这种形式的文档：

```
{
                "_id" : ObjectId("509d27069cc1ae293b36928d"),
                "comments" : [ {
                        "subject" : "Sample Comment 1 ",
                        "body" : "Comment1 Body.",
                        "author_id" : "author2",
```

```
                    "created_date" :
                ISODate("2015-07-06T13:34:23.929Z")}...]
    }
            "_id" : ObjectId("509d27069cc1ae293b36928d"),
            "comments" : [
                    {
                            "subject" : "Sample Comment 2",
                            "body" : "Comments Body.",
                            "author_id" : "author2",
                            "created_date" :
                        ISODate("2015-07-06T13:34:23.929Z")
                    }...]}
```

此方法主要的缺点是，你得到的数据会远远多于你实际需要的数据。实际上，你无法要求只取出 author2 的评论；你必须取得 author2 评论过的博文，其中也包括那些博文上的所有其他评论。这些数据需要在应用程序代码中进一步过滤。

另一方面，假定你决定使用一个标准化模式。在这个例子中将有 3 个文档：“Authors”、“Posts”和“Comments”。

“Authors”文档将具有特定于作者的内容，比如 Name、Age、Gender 等，而“Posts”文档将具有特定于博文的详细信息，比如博文创建信息、博文作者、实际内容以及博文的标题。

“Comments”文档将具有博文的评论，比如 CommentedOn 日期时间、创建的作者以及评论的内容。这被描述如下：

```
// Authors document:
{
"_id": ObjectId("508d280e9cc1ae293b36928e "),
"name": "Jenny",
..........
}
//Posts Document
        {
                "_id" : ObjectId("508d27069cc1ae293b36928d"),
    ..................
        }
// Comments document:
{
"_id": ObjectId("508d359a9cc1ae293b3692a0"),
"Author": ObjectId("508d27069cc1ae293b36928d"),
"created_date" : ISODate("2015-07-06T13:34:59.336Z"),
"Post_id": ObjectId("508d27069cc1ae293b36928d"),
..........
}
```

在此场景中，找出由“author2”发表的评论这个查询可以通过 comments 集合上的

一个简单 find()来实现：

```
db.comments.find({"author": "author2"})
```

一般来说，如果应用程序的查询模式为众所周知，且数据往往只能通过一种方式来访问，那么嵌入的方法将很好地发挥作用。相反，如果应用程序可以用许多不同的方式查询数据，或者你无法预料到数据可能被查询的模式，那么一种更为“标准化”的方法可能会更好。

例如，在上述模式中，将能够对评论排序或者使用 limit、skip 操作符返回一个更受限制的评论集。在嵌入的例子中，无法摆脱按照其在博文中存储的顺序来检索所有评论的环节。

另一个权衡是否使用文档引用的因素就是，具有一对多关系的情况。

例如，一个有大量读者参与的受欢迎的博客中，可能某篇博文就有数百甚或数千评论。在这种情况下，嵌入就会带来显著的负面影响：

- **读取性能上的影响：**随着文档大小的增长，它将消耗更多的内存。内存方面的问题在于，MongoDB 数据库会在内存中缓存频繁访问的文档，而这些文档变得越大，它们适合放入内存中的可能性就越小。这将导致在检索文档时出现更多的页面错误，也将导致随机磁盘 I/O，进而降低性能。
- **更新性能上的影响：**随着文档大小的增长，以及在这样的文档上执行更新操作以附加数据，最终 MongoDB 会需要将该文档移动到具有更多可用空间的区域。当这种情况出现时，这种移动将显著降低更新性能。

除此之外，MongoDB 文档具有 16MB 这一固定的大小限制。尽管这是要意识到的一个情况，但将经常遇到由于内存压力以及在到达 16MB 大小限制之前文档的复制很顺畅而产生的问题。

权衡是否使用文档引用的一个最终因素是多对多或 M:N 关系的情况。

例如，在上面的示例中，存在标签。每篇博文都可以具有多个标签并且每个标签都可以关联到多个博文条目。

实现博文-标签 M:N 关系的一种方法是使用以下三个集合：

- Tags 集合，它将存储标签详细信息
- Blogs 集合，它将存储博文详细信息
- 第三个集合，被称为 Tag-To-Blog Mapping，它将在标签和博文之间进行映射

此方法类似于关系型数据库中的方法，但这不会影响应用程序的性能，因为查询最终将进行大量应用程序级别的“联结”。

或者，可以在博文文档中使用嵌入了标签的嵌入模型，但这将会导致数据重复。尽管这对读取操作有一些简化，但它将增加更新操作的复杂性，因为在更新一个标签详细信息时，用户需要确保在标签被嵌入到其他博文文档的每一个地方都对该标签进行更新。

因此，对于多对多联结来说，一个折中的办法通常是最好的，嵌入一组_id 值而非完整的文档：

```
// Tags document:
```

```
{
"_id": ObjectId("508d35349cc1ae293b369299"),
"TagName": "Tag1",
.........
}
// Posts document with Tag IDs added as References
//Posts Document
        {                  "_id" : ObjectId("508d27069cc1ae293b36928d"),
                "tags" : [
                        ObjectId("509d35349cc1ae293b369299"),
                        ObjectId("509d35349cc1ae293b36929a"),
                        ObjectId("509d35349cc1ae293b36929b"),
                        ObjectId("509d35349cc1ae293b36929c")
                ],.....................................
        }
```

尽管查询会有一点复杂，但你不再需要担心在每个地方更新标签。

总的来说，MongoDB 中的模式设计是你需要很早就做出的决定之一，并且它取决于应用程序的需求和查询。

如你所见，当需要将数据放在一起进行访问或者你需要进行原子更新时，嵌入将产生积极影响。不过，如果在查询时需要更多的灵活性或者如果拥有一种多对多关系，那么使用引用将是一个好的选择。

最终，决策取决于你的应用程序的访问模式，并且 MongoDB 中没有固定不变的规则。在下一节中，将学习与各种数据建模考虑事项有关的内容。

3. 数据建模的决策

这涉及决定如何构造文档，以便有效建模数据。要决策的重要一点在于，你是否需要嵌入数据或者使用对数据的引用(例如，是否使用嵌入或引用)。

有个例子可以很好地揭示这一点。假定你有一个书籍评论站点，其中有作者和书籍以及具有嵌套评论的评价系统。

现在问题在于，如何构造这些集合。该决策取决于对每本书籍所预期的评论数量以及将要执行的读取以及写入操作的频率是多少。

4. 操作上的注意事项

除了元素彼此交互的方式之外(比如，是否以嵌入方式存储文档或者使用引用)，在为应用程序设计一个数据模型时，还有若干其他操作上的因素也很重要。后面几节中将介绍这些因素。

数据生命周期管理

如果应用程序具有只需要在有限的一段时间内在数据库中持久化的数据集，那么就

需要此功能。

假设你需要将与评价和评论有关的数据保留一个月，那么就可以考虑使用此功能。

这是通过使用集合的生存周期(Time to Live，TTL)功能来实现的。集合的 TTL 功能会确保文档在一段时间后过期。

此外，如果应用程序需求是仅处理最近插入的文档，那么使用固定集合将有助于优化性能。

索引

可以创建索引来支持经常用到的查询，以便提高性能。默认情况下，MongoDB 会在_id 字段上创建一个索引。

下面是在创建索引时需要考虑的一些要点：

- 每个索引至少需要 8KB 的数据空间。
- 对于写操作，索引的添加将带来一些不良的性能影响。因此，对于具有大量写入的集合来说，使用索引的代价可能会很大，因为对于每一个插入，都必须将键添加到所有的索引。
- 索引适用于具有大量读取操作的集合，比如读与写的操作比例很高的那些集合。未被索引的读操作不会受到索引的影响。

分片

在设计应用程序模型时，一个重要的因素是，是否要对数据进行分区。这是通过在 MongoDB 中使用分片来实现的。

分片也被称为数据分区。在 MongoDB 中，一个集合是通过其跨机器群集分布的文档来分区的，这些文档就被称为碎片。这样会对性能产生显著的影响。我们将在第 7 章中探讨更多与分片有关的内容。

大量的集合

对比在单个集合中存储数据，下面是具有多个集合时的设计考虑事项：

- 选择多个集合用于数据存储是没有性能损失的。
- 在高吞吐量的批处理应用程序中，为不同的数据类型使用单独的集合会提高性能。

当在设计具有大量集合的模型时，你需要考虑以下行为：

- 每个集合都会关联数千字节的特定最小开销。
- 每个索引至少需要 8KB 的数据空间，其中包括_id 索引。

现在你知道了每个数据库的元数据都被存储在<database>.ns 文件中。每个集合与索引在命名空间文件中都有其自己的条目，因此在决定实现大量集合时你需要考虑命名空间文件的大小限制。

文档的增长

有一些更新，比如将一个元素推送到一个数组中、添加新字段等，会造成文档大小的增长，这将导致文档从一个位置移动到另一个位置，以便适应该文档。文档迁移的这一过程会同时带来资源和时间方面的消耗。尽管 MongoDB 提供了填充来最小化迁移的发生，但你可能还是需要手动处理文档增长。

6.4 本章小结

在本章中，你学习了基础的 CRUD 操作以及高级的查询功能。你还探究了存储和检索数据的两种方式：嵌入和引用。

在下一章中，将学习与 MongoDB 架构、其核心组件以及功能有关的知识。

第 7 章

MongoDB 架构

“MongoDB 架构涵盖了 MongoDB 的深层架构概念。”

在本章中，将学习与 MongoDB 架构有关的知识，尤其是核心程序与工具、独立部署、分片概念、复制概念以及生产环境部署。

7.1 核心程序

MongoDB 包中的核心组件是：

- mongod，它是核心数据库程序
- mongos，它是用于分片群集的控制器和查询路由器
- mongo，它是交互式 MongoDB shell

这些组件在 bin 文件夹下是作为应用程序来提供的。我们将详细探讨这些组件。

7.1.1 mongod

MongoDB 系统中的主守护程序被称为 mongod。此守护程序会处理所有的数据请求、管理数据格式并且执行用于后台管理的操作。

当一个 mongod 在无任何参数的情况下运行时，它会连接到默认的数据目录，也就是 C:\data\db 或/data/db，以及默认的端口 27017，它会在这个端口上侦听 socket 连接。

重要的是在 mongod 程序启动时确保该数据目录存在，并且你拥有对该目录的写权限。

如果该目录不存在或者你对该目录没有写权限，那么这个程序的启动将会失败。如果默认端口 27017 不可用，则该服务器将启动失败。

mongod 还有一个 HTTP 服务器，它会侦听比默认端口大 1000 的端口，因此，如果启动使用默认端口 27017 的 mongod，那么在这种情况下，该 HTTP 服务器将位于端口 28017 上并且可以使用 URL http://localhost:28017 来访问。此基础 HTTP 服务器提供了与数据库有关的管理运营信息。

7.1.2 mongo

mongo 为开发人员提供了一个交互式 JavaScript 接口以便在数据库上直接测试查询

和操作，并且也可用于系统管理员来管理数据库。这完全是通过命令行来完成的。当 mongo shell 启动后，它将连接到被称为 test 的默认数据库。这个数据库连接值被分配到全局变量 db。

作为开发人员或管理员，需要在完成第一次连接之后将数据库从 test 修改为你的数据库。可以通过使用<databasename>来达成该目标。

7.1.3 mongos

mongos 被用于 MongoDB 分片。它充当了一种路由服务，会处理来自应用层的查询并且判定所请求的数据位于分片群集的哪个位置。

我们将在 7.5 节“分片”中更为详尽地探讨 mongos。现在可以将 mongos 看作将查询路由到保存数据的正确服务器的程序。

7.2 MongoDB 工具

除了核心服务之外，还有各种工具是作为 MongoDB 安装的一部分来提供的：

- **mongodump**：此工具被用作一种有效备份策略的一部分。它会创建数据库内容的一份二进制导出。
- **mongorestore**：由 mongodump 工具创建的二进制数据库转储是使用 mongorestore 工具导入到一个新的或者现有的数据库中的。
- **bsondump**：这一工具会将 BSON 文件转换成可供人们阅读的格式，比如 JSON 或 CSV。例如，这个工具可被用于读取 mongodump 生成的输出文件。
- **mongoimport、mongoexport**：mongoimport 提供了一种以 JSON、CSV 或 TSV 格式获取数据并且将这些数据导入到一个 mongod 实例中的方法。mongoexport 提供了一种方法来将数据从 mongod 实例中导出成 JSON、CSV 或 TSV 格式。
- **mongostat、mongotop、mongosniff**：这些工具提供了与 mongod 实例的当前操作有关的诊断信息。

7.3 独立部署

独立部署被用于开发目的；它不保证冗余数据并且它也无法保障在失败的情况下进行恢复。因此不建议将它用在生产环境中。独立部署具有以下组件：单个 mongod 以及连接到该 mongod 的一个客户端，如图 7-1 所示。

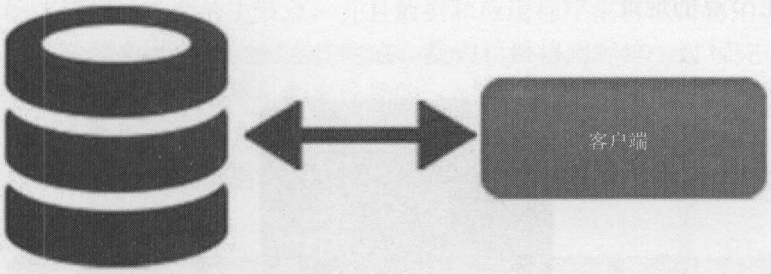

图 7-1 独立部署

MongoDB 使用了分片和复制功能通过分发和复制数据来提供高可用的系统。在接下来的几节中，将介绍分片和复制。通过这几节内容的学习，将了解到推荐的生产环境部署架构。

7.4 复制

在一个独立部署中，如果 mongod 不可用，那么将面临所有数据丢失的风险，这对于生产环境来说是不可接受的。复制被用于提供针对这类数据丢失的安全保障。

通过将数据复制到不同的节点上，复制提供了数据冗余，因而也就为节点故障的情况提供了数据保护。在 MongoDB 部署中，复制提供了高可用性。

复制还简化了某些管理任务，因此像备份这样的例行任务就可以被转移到复制副本，让主副本可以解脱出来应对重要的应用程序请求。

在某些应用场景中，通过让客户端读取不同的数据副本，它还可以有助于扩展读取规模。

在本节中，将学习复制在 MongoDB 中是如何工作的以及其各种组件。MongoDB 支持两种复制类型：传统的主/从复制和副本集。

7.4.1 主/从复制

在 MongoDB 中，传统的主/从复制是可用的，但仅推荐将其用于超过 50 个节点的复制。首选的复制方法是副本集，我们稍后将阐释这一点。在主/从复制类型中，有一个主节点以及从这个主节点复制数据的若干从节点。使用这一复制类型的唯一优势在于，在一个群集中并没有对从节点的数量进行限制。然而，数千个从节点将会让主节点过载，因此在实际的场景中，最好不要超过十几个从节点。此外，这一复制类型不会自动故障转移并且提供了较少的冗余。

在基础的主/从设置中，有两种 mongod 实例的类型：一个实例位于主模式中，而另一个位于从模式中，如图 7-2 所示。由于从节点是从主节点复制的，因此所有的从节点

都需要知道主节点的地址。

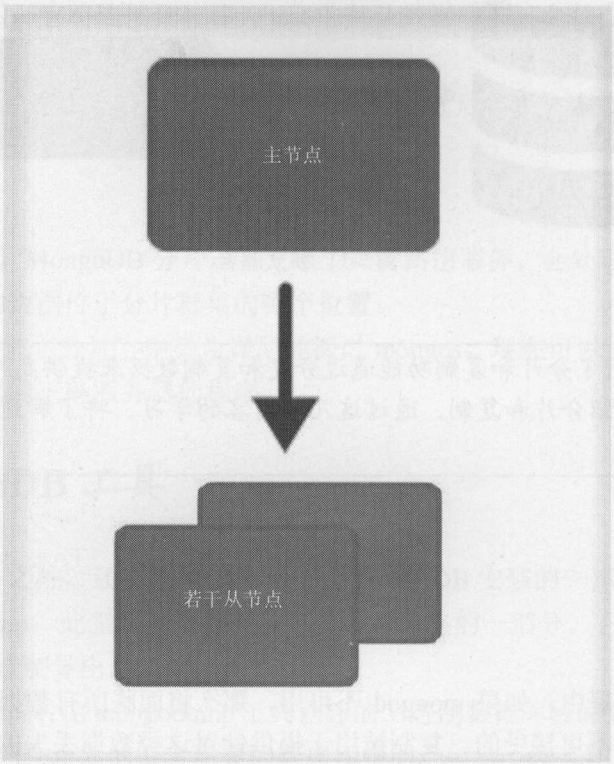

图 7-2 主/从复制

主节点会存留一个固定集合(oplog)，这个集合存储了对数据库进行的逻辑写入的顺序历史。

从节点会使用这个 oplog 集合来复制数据。由于这个 oplog 是一个固定集合，因此如果从节点的状态远远落后于主节点的状态的话，则该从节点可能就不会同步。在此场景下，复制将停止并且需要手动干预来重新恢复该复制。

一个从节点不同步的背后有两个主要原因：

- 从节点关闭或停止并且稍后重启。在这段时间内，oplog 可能会删除需要被应用到该从节点上的操作日志。
- 从节点在执行来自主节点的可用更新时很慢。

7.4.2 副本集

副本集是传统主/从复制的一种复杂形式，并且是 MongoDB 部署中的一种推荐方法。

副本集实质上是主/从复制的一种类型，但它们提供了自动化故障转移。在副本集上下文中，副本集有一个主节点，它被称为主副本，还有多个从节点，它们被称为辅助副本；不过，不同于主/从复制的是，副本集中没有一个节点是固定的主副本。

如果副本集中的一个主节点关闭了，那么其中一个从节点就会被自动提升为主节点。

客户端会开始连接到这个新的主节点，并且数据和应用程序都将保持可用。在一个副本集中，这一故障转移会以自动化的方式完成。我们稍后将阐释这一过程完成的详情。

主副本节点是通过一种选举机制来选择的。如果主副本关闭了，那么所选择的节点将被选中为主副本节点。

图 7-3 显示了一个双成员副本集的故障转移是如何完成的。我们来探讨双成员副本集在故障转移方面的完成步骤。

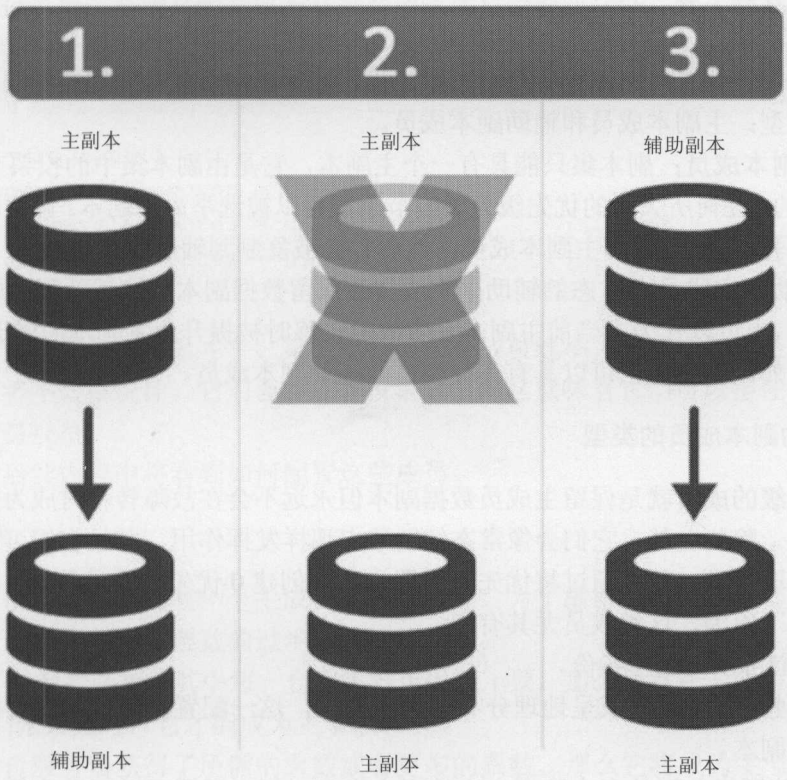

图 7-3 双成员副本集故障转移

(1) 主副本关闭，辅助副本被提升为主副本。

(2) 最初的主副本恢复，它会充当从节点，并且变成辅助副本节点。

要注意的要点是：

- 副本集是 mongod 的群集，它会在彼此之间进行复制，并且确保自动化故障恢复。
- 在副本集中，一个 mongod 将会是主副本成员，而其他的将是辅助副本成员。
- 主副本成员是由副本集的成员选举出来的。所有的写入都会被指向该主副本成员，而辅助副本成员会从主副本异步使用 oplog 进行复制。
- 辅助副本的数据集会反射主副本数据集，让它们可以在当前主副本不可用的情况下被提升为主副本。

副本集复制在成员数量上有一个限制。在版本 3.0 之前，该限制为 12，但在版本 3.0 中这个限制被更改为 50。因此现在副本集复制只能拥有最多 50 个成员，并且在 50 个成

员的副本集中，在任意指定时点，只有 7 个成员可以参与一次投票。我们将详细地阐释副本集中的投票概念。

从版本 3.0 开始，副本集成员可以使用不同的存储引擎。例如，WiredTiger 存储引擎可用于辅助副本成员，而 MMAPv1 引擎可用于主副本。在后面几节中，将介绍可以与 MongoDB 一起使用的不同存储引擎。

---

1. 主副本和辅助副本成员

在继续了解副本集如何发挥作用之前，我们来看看副本集可以拥有的成员类型。有两种成员类型：主副本成员和辅助副本成员。

- **主副本成员**：副本集只能具有一个主副本，它是由副本集中的投票节点选举出来的。任何所关联的优先级为 1 的节点都可以被选举为主副本。客户端会将所有的写操作重定向到主副本成员，然后它会被复制到辅助副本成员。
- **辅助副本成员**：常态的辅助副本成员会保留数据副本。该辅助副本成员可以投票，也可以作为在当前主副本发生故障转移时被提升为主副本的候选。

除此之外，副本集也可以具有其他类型的辅助副本成员。

2. 辅助副本成员的类型

0 优先级的成员就是保留主成员数据副本但永远不会在故障转移时成为主副本的辅助副本成员。除此之外，它们会像常态辅助节点那样发挥作用，并且它们可以参与投票以及接收读取请求。可以通过将优先级设置为 0 来创建 0 优先级成员。

出于以下原因，这类成员尤其有用：

(1) 它们可以充当冷备份。

(2) 在使用各种硬件或呈地理分布的副本集中，这一配置会确保只有合格的成员能被选举为主副本。

(3) 在跨网络分区的多个数据中心的副本集中，这一配置可以帮助确保主数据中心具有合格的主副本。这被用于确保快速的故障转移。

隐藏成员就是客户端应用程序不可见的 0 优先级成员。就像 0 优先级成员一样，这一成员也会保留主成员的数据副本，无法成为主副本，并且可以参与投票，但不同于 0 优先级成员，它无法服务于任何读取请求或者接收超出复制需求内容的任何通信。可以通过将 hidden 属性设置为 true 来将一个节点设置为隐藏成员。在一个副本集中，这些成员可以专用于报告需求或者备份。

延迟成员就是从主副本的 oplog 延迟复制数据的辅助副本成员。这有助于从人为错误中进行恢复，比如意外删除数据库或者由于不成功的应用程序更新导致的错误。

在决定延迟时间时，可以考虑你的维护周期以及 oplog 的大小。延迟时间应该等于或者大于维护窗口并且应该以确保复制期间没有操作丢失的方式来设置 oplog 大小。

注意，由于延迟成员不会像主节点那样具有更新的数据，因此其优先级应该被设置为 0，这样它们就无法变成主副本。另外，其 hidden 属性应该是 true，以避免任何读取

请求。

仲裁者就是不持有主节点数据副本的辅助成员，因此它们绝不会成为主节点。它们仅被用作参与投票的成员。这使得副本集可以具有非偶数的节点数量，而无需任何会带来数据复制的复制开销。

非投票成员会持有主节点的数据副本，它们可以接受客户端读取操作，并且它们也可以变成主副本，但它们无法在选举中投票。

可以通过设置一个成员的选票为 0 来禁用该成员的投票功能。默认情况下，每一个成员都可以投一次票。假设你有一个具有 7 个成员的副本集。在 mongo shell 中使用以下命令，就可以将第 4 个、第 5 个和第 6 个成员的选票设置为 0：

```
cfg_1 = rs.conf()
cfg_1.members[3].votes = 0
cfg_1.members[4].votes = 0
cfg_1.members[5].votes = 0
rs.reconfig(cfg_1)
```

尽管这一设置允许将第 4 个、第 5 个和第 6 个成员选举为主节点，但是在投票时，它们的选票将不会被统计。它们会变成非投票成员，这意味着它们可以担任候选人，但无法对其自身投票。

在本章后续内容中将看到如何配置这些成员。

3. 选举

在本节中，将了解选取一个主成员的选举过程。为了赢得选举，服务器不仅需要获得多数票，还需要获得总票数的过半票数才行。

如果有 X 台服务器，其中每一台服务器可以投 1 票，那么只有在一台服务器获得至少[(X/2) + 1]张票数时，它才能成为主节点。

如果一台服务器获得了所需的票数或者更多的票数，那么它将成为主节点。

关闭的主节点仍然是集合中的一部分；当它恢复开启时，它将充当辅助服务器，直到它再次获得多数票。

这类投票系统的复杂之处在于，你无法仅用两个节点来充当主节点和从节点。在这种使用场景下，你总计将只有两票，而要成为一个主节点，一个节点需要多数票，在这种情况下也就是全部的两票。如果其中一台服务器关闭了，那么另一台服务器最终将只有两票中的一票，因而它将永远不会被提升为主节点，因此它将仍然是从节点。

在网络分区的情况下，主节点将失去多数票，因为它将只有自己的一票，并且它将被降级为从节点，而充当从节点的那个节点也仍将是从节点，因为它也缺少多数票。你最终将有两个从节点，直到这两台服务器再次彼此互通为止。

副本集有若干方式可以避免这样的情况出现。最简单的方式就是使用一个仲裁者来帮助解决这样的冲突。它是非常轻量级的并且只是一个投票者，因此它本身可以运行在其中一台服务器上。

我们现在来看看上述场景在使用一个仲裁者的情况下又会如何。我们首先考虑网络

分区的情况。如果有一个主节点、一个从节点和一个仲裁者，它们都有各自的一票，总计 3 票。如果出现主节点和仲裁者位于一个数据中心而从节点位于另一个数据中心的网络分区，那么该主节点将仍然是主节点，因为它将仍然拥有多数票。

如果主节点在没有网络分区的情况下出现故障，那么从节点就可以被提升为主节点，因为它将拥有两张选票(从节点+仲裁者)。

这种三台服务器的设置提供了一种健壮的可进行故障转移的部署。

示例——选举过程运行的更多详情

本节将阐释选举是如何实现的。

假设有一个副本集，它具有以下 3 个成员：A1、B1 和 C1。每一个成员每几秒都会与其他成员交互心跳请求。这些成员会为这些请求返回其当前情况信息。A1 会发送心跳请求给 B1 和 C1。B1 和 C1 会返回其当前情况信息，比如它们当前的状态(主节点或辅助节点)、它们当前的时钟、它们是否有资格被提升为主节点等。A1 会接收到所有这些信息并且更新其集合的“映射”，它会保存这些信息，比如成员变更后的状态、已经关闭或启动的成员，以及通信往返时间。

在更新时，A1 的映射会变更，它将检查一些内容，这取决于其状态：

- 如果 A1 是主节点，并且其中一个成员已经关闭，那么它将确保它自身仍旧能够获得该集合的多数票。如果无法达成该目的，那么它将让其自身降级为辅助状态。

  降级：当 A1 经历降级时，会有一个问题。在 MongoDB 中，默认情况下写操作是即发即弃的(例如，客户端发出写请求，但不会等待响应)。如果一个应用程序在主节点正关闭时执行默认的写操作，那么它将永远不会知道这些写操作实际上没有发生，并且最终可能会丢失数据。因此建议使用安全的写操作。在这一场景中，当主节点正关闭时，它会关闭其所有客户端的连接，这样就会产生套接字错误返回给客户端。然后客户端库需要重新检查新的主节点是谁，从而避免丢失其写操作的数据。
- 如果 A1 是辅助节点，并且如果其映射并没有变更，那么它将间或检查它是否选举了自己。

  A1 要做的第一个任务就是运行一次健全性检查，其中它会检查一些问题的答案，比如，A1 已经将它自己视作主节点了吗？另一个成员将它自己视作主节点了吗？A1 是否不适合当选？如果它无法回答所有这些基础问题，那么 A1 将继续原样保持闲置；否则，它将继续处理选举过程：

  - A1 会给集合的其他成员发送一条消息，这个例子中就是 B1 和 C1，表明“我计划变成一个主节点。请推荐”。
  - 当 B1 和 C1 接收到这条消息时，它们将检查围绕它们的视图。它们将运行一长串健壮性检查，比如，是否有任何其他节点可以成为主节点？A1 是否具有最近的数据或者有没有任何其他节点有最近的数据？如果所有的检查看起来没问题，那么它们就会发送一条“请继续”的消息；不过，如果任何一个检查失败，则会发送一条“停止选举”的消息。

- 如果任何一个成员发送一条“停止选举”的回复，那么该选举将被取消并且 A1 仍旧是一个辅助成员。
- 如果从所有节点都接收到“请继续”，那么 A1 将前进到选举过程的最终阶段。

在第二个(最终)阶段，

- A1 会向其余的成员发送一条消息，声明其参选资格。
- 成员 B1 和 C1 会进行一次最终检查来确保所有的答案仍旧像之前那样保持为 true。
- 如果确实如此，则 A1 将被允许使用其选举锁，这会阻止其投票功能 30 秒并且回送一张选票。
- 如果任何检查没有保持为 true，则会发送一张否决票。
- 如果接收到任何一张否决票，则选举将停止。
- 如果没有节点投否决票并且 A1 获得了多数选票，那么它将变成一个主节点。

选举会受到优先级设置的影响。0 优先级成员绝不会成为一个主节点。

4. 数据复制过程

我们来看看数据复制如何工作。副本集的成员会连续不断地复制数据。每一个成员，包括主副本成员在内，都会维护一个 oplog。oplog 是一个固定集合，其中成员会维护所有操作的一份记录，这些操作都是在数据集上执行的。

辅助副本成员会复制主副本成员的 oplog 并且以异步方式应用所有的操作。

oplog

oplog 代表操作日志。oplog 就是一个固定集合，其中记录了所有修改数据的操作。

oplog 是在一个特殊的数据库中维护的，即它位于集合 oplog.$main 中。每一个操作都被作为一个文档来维护，其中每一个文档都对应一个在主服务器上执行的操作。文档包含各种键，其中包括以下键：

- **ts**：这会存储执行操作时的时间戳。它是内部类型并且由 4 个字节的时间戳和 4 个字节的递增计数器构成。
- **op**：这会存储关于所执行的操作类型的信息。其值被存储为 1 个字节的代码(比如，它将为一个插入操作存储一个“I”)。
- **ns**：这个键会存储被执行操作的集合的命名空间。
- **o**：这个键会指定被执行的操作。比如插入，这个键将存储文档以便插入。

只有修改数据的操作才会在 oplog 中维护，因为它是确保辅助节点数据与主节点数据保持同步的一种机制。

oplog 中存储的操作会被转换，这样它们就会保持幂等，也就是说即使它被应用到辅

助节点多次，辅助节点数据也仍将保持一致。由于 oplog 是一个固定集合，会添加每一个新的操作，因此最老的操作将被自动移除。这样做是为了确保它不会超出预设定的边界，也就是 oplog 大小。

无论副本集成员何时首次启动，MongoDB 都会依据 OS 的不同根据一个默认大小来创建 oplog。

在 MongoDB 中，默认情况下可用空间或 5%的空间会被用于 Windows 和 64 位 Linux 实例上的 oplog。如果其大小低于 1GB，那么 MongoDB 就会分配 1GB 的空间。

尽管大多数情况下默认大小就足够了，但可以在启动服务器时使用–oplogsize 选项来指定以 MB 为单位的 oplog 大小。

如果有以下工作负荷，那么可能就需要考虑 oplog 的大小：

- **同时对多个文档进行更新：**由于这些操作需要被转换成幂等操作，因此这一场景可能最终需要很大的 oplog 大小。
- **以相同速度发生的涉及等量数据的删除和插入：**在这种场景下，尽管数据库大小不会增加，但这些操作被转换成幂等操作将产生较大的 oplog。
- **大量的就地更新：**尽管这些更新不会改变数据库大小，但在 oplog 中将这些更新记录为幂等操作会产生较大的 oplog。

初始化同步与复制

当成员处于以下两种情况之一时，就会完成初始化同步：

(1) 节点首次启动(比如，它是一个新节点并且没有数据)。

(2) 节点变得过时，其中主节点已经重写了 oplog，而该节点并没有复制数据。在这种情况下，数据将被移除。

在这两种情况下，初始化同步都涉及以下步骤：

(1) 首先，所有的数据库都会被克隆。

(2) 使用源节点的 oplog，变更会被应用到其数据集。

(3) 最后，会在所有的集合上构建索引。

在初始化同步之后，副本集成员会持续复制变更，以便保持最新状态。

大多数同步都产生于主节点，但在同步仅产生于一个辅助节点时，就会启用链式复制(比如，根据 ping 时长和其他成员复制的状态，同步目标改变了)。

同步——正常操作

在正常操作中，辅助节点会选择一个成员来同步其数据，然后会从选中的源的 oplog 集合(local.oplog.rs)中拉取操作。

一旦得到了操作(op)，该辅助节点就会执行以下步骤：

(1) 它首先会将 op 应用到其数据副本。

(2) 然后它会将该 op 写入到其本地 oplog。

(3) 一旦该 op 被写入到 oplog，它就会请求下一个 op。

假定它在步骤 1 和步骤 2 之间失败了，那么它就会再次重头执行这些步骤。在这种情况下，它会假设该操作还没有被执行并且重新应用该操作。

由于 oplog 操作是幂等的，因此相同的操作可以被应用任意次数，并且每一次其结果文档都将相同。

如果有以下文档

{I:11}

对相同的文档执行一个递增操作，比如

对主节点执行{$inc:{I:1}}

这样的话，以下内容将被存储到主节点 oplog 中：

{I:12}。

这会被辅助节点复制。因此，即使该日志被应用多次，其值仍旧会是相同的。

启动

当一个节点被启动时，它会检查其本地集合来找出 lastOpTimeWritten。这是在该辅助节点上被应用的最近一个 op 的时间。

可以使用以下 shell 帮助命令在 shell 中找到最近的 op：

```
>rs.debug.getLastOpWritten()
```

该输出结果会返回一个名称为 ts 的字段，它会描述最近的 op 时间。

如果一个成员启动并且找到 ts 条目，那么它首先会选择一个目标来从其进行同步并且它会将同步作为一个正常操作来启动。不过，如果没有找到条目，那么该节点就会开始初始同步过程。

从何处同步

在本节中，将了解如何选择要同步的源。从 2.0 开始，服务器会基于平均 ping 时长自动从“最近的”节点进行同步。

当启用一个新节点时，它会向所有的节点发送心跳并且监控响应时间。根据所接收到的数据，之后它会使用以下算法来判定要从哪个成员进行同步：

```
循环每一个健康的成员：
   如果状态是主节点
      则将该成员添加到可能的同步目标集
   如果成员的 lastOpTimeWritten 比本地 lastOpTimeWritten 大
则将该成员添加到可能的同步目标集
设置 sync_from = MIN (同步目标集成员的 PING 时长)
```

提示：“健康成员”可以被视作“正常”主成员或辅助成员。

在版本 2.0 中，从副本的延迟节点被争议性地纳入了“健康”节点中。从版本 2.2 开始，延迟节点和隐藏节点都被排除在“健康”节点之外了。

运行以下命令将显示被选中作为同步源的服务器：

```
db.adminCommand({replSetGetStatus:1})
```

syncingTo 的输出字段仅出现在辅助节点上，并且会提供它正在从哪个节点同步的信息。

让写操作应用于链式从节点

你已经看到了，上述用于选择一个源来进行同步的算法表明，从节点链接是半自动的。在启动一台服务器时，它很大可能会从相同数据中心选择一台服务器来同步，因而会降低 WAN 的流量。

不过，这不会导致循环，因为节点只会从具有一个大于其自身的 lastOpTimeWritten 最近值的辅助节点进行同步。你不会遇到这样的场景，其中 N1 从 N2 同步，而 N2 从 N1 同步。永远只会出现 N1 从 N2 同步或者 N2 从 N1 同步的这两种情况中的一种。

在本节中，将看到 w(写操作)如何应用于从节点链接。如果 N1 从 N2 同步，而 N2 进一步从 N3 同步，在这种情况下，N3 如何知道何时 N1 才会被同步到。

当 N1 开始从 N2 同步时，会发送一条特殊的“握手”消息，它会提示 N2，N1 将从其 oplog 同步。由于 N2 不是主节点，因此它会将该消息转发给它所同步的节点(比如，它会假装 N1 打开一个到 N3 的连接)。上述步骤结束时，N2 就打开了到 N3 的两个连接：一个连接用于其自身，另一个用于 N1。

无论何时由 N1 到 N2 发出了一个 op 请求，该 op 都会由 N2 从其 oplog 发出，并且一个伪请求会在 N1 到 N3 的链接上被转发，如图 7-4 所示。

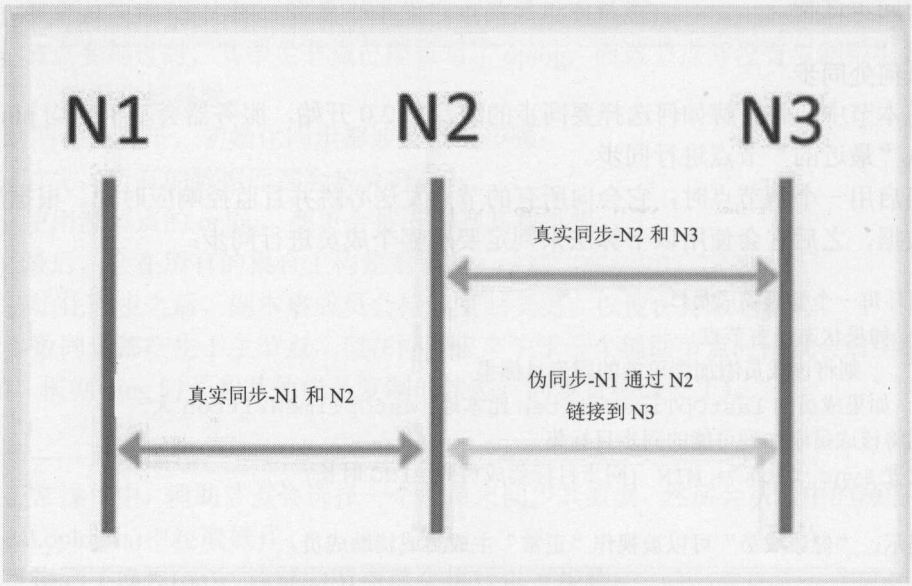

图 7-4 通过链式从节点进行写入

尽管这会最小化网络流量，但它增加了写操作到达所有成员的绝对时长。

5. 故障转移

在本节中，将了解在副本集中是如何处理主成员和辅助成员的故障转移的。副本集

的所有成员都彼此连接。如图 7-5 所示，它们会在相互之间交换心跳消息。

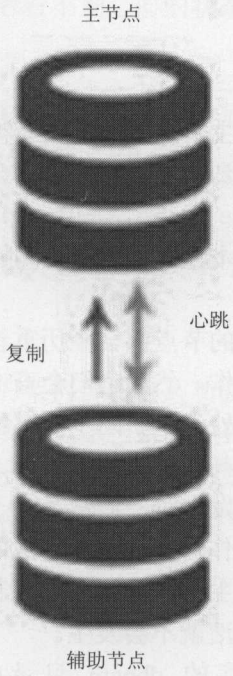

图 7-5　心跳消息交换

因此一个缺少心跳的节点会被视作故障节点。

如果该节点是辅助节点

如果该节点是一个辅助节点，那么将会从副本集的成员中移除它。以后，当它恢复时，它可以重新加入。在它重新加入之后，它需要更新最近的变更。

(1) 如果故障的时间很短，那么它会连接到主节点并且获取最近的更新。

(2) 不过，如果故障时间很长，那么该辅助服务器就需要重新与主节点同步，其中它会删除其所有数据并且执行初始化同步，就像它是新服务器那样。

如果该节点是主节点

如果该节点是一个主节点，在这种情况下，如果原始副本集的大多数成员都能够彼此连接，那么就会由这些节点选举一个新的主节点，这与副本集的自动故障转移功能一致。

该选举过程将会由任意无法成为主节点的节点初始化。

这个新的主节点是由大多数副本集节点选举出来的。仲裁者可被用于在应用场景中打破链接，比如在网络分区将参与节点划分为两半并且无法达到多数票的场景。

具有最高优先级的节点将会是新的主节点。如果有多个相同优先级的节点，那么数据的实时性可被用于打破链接。

主节点使用一个心跳来追踪有多少节点对它可见。如果可见节点的数量小于多数票

的数量，则主节点会自动回退成辅助状态。这一场景会避免主节点在被网络分区分隔时发挥作用。

6. 回滚

在主节点变更的情况下，新的主节点上的数据会被假定为系统中最新的数据。当之前的主节点联结回来时，任何应用到其上的操作也将会被回滚。然后它将与新的主节点进行同步。

回滚操作会将还没有在整个副本集复制的所有写操作还原。这样做是为了维持整个副本集的数据库一致性。

在连接到新的主节点时，所有的节点都会经历重新同步的过程，以确保回滚完成。节点会遍历新的主节点上没有的操作，然后它们会查询新的主节点以返回受到这些操作影响的文档的最新副本。这些节点处于重新同步的过程中并且表明正在恢复中；直到该过程完成之前，它们都没有资格参与主节点选举。

这种情况很少发生，不过当它发生时，通常是由于网络分区造成复制延迟，其中有辅助接点无法跟上之前主节点上操作的吞吐节奏造成的。

需要注意的是，如果写操作在主节点关闭之前复制到其他成员，并且那些成员可以被副本集的多数节点访问，那么回滚就不会发生。

回滚数据会被写入到数据库的 dbpath 目录中带有像<database>.<collection>.<timestamp>.bson 这样的文件名的 BSON 文件中。

管理员可以决定忽略或者应用回滚数据。只有在所有节点都与新的主节点保持同步并且已经回滚到一个一致状态时才能应用回滚数据。

可以使用 Bsondump 来读取回滚文件的内容，然后需要使用 mongorestore 手动将这些内容应用到新的主节点。

MongoDB 没有提供方法来自动处理回滚的情况。因此需要人工介入来应用回滚数据。在应用回滚时，至关重要的是确保这些回滚内容被复制到副本集的所有成员或者至少复制到副本集的一些成员，这样才能避免任何故障转移回滚的情况出现。

7. 一致性

你已经看到了，副本集成员通过读取 oplog 来保持彼此之间的数据复制。那么数据的一致性是如何维护的呢？在本节中，将介绍 MongoDB 如何确保总是访问一致的数据。

在 MongoDB 中，尽管读取可以被路由到辅助节点，但写入总是会被路由到主节点，这就根除了两个节点同时尝试更新相同数据集的情况。主节点上的数据集总是一致的。

如果读取请求被路由到了主节点，那么它将总是会看到最新的变更，这意味着读取操作总是会与最近的写操作保持一致。

不过，如果应用程序已经将读取偏好变更为从辅助节点读取的话，那么用户可能就有一定概率看不到最近的变更或者看不到之前的状态。这是因为写操作在辅助节点上是异步复制的。

此行为被称为最终一致性，它意味着尽管辅助节点的状态与主节点状态并不一致，但随着时间的推移，它最终会变得一致。

没有办法确保从辅助节点的读取保持一致，除非发出写关注来确保在写操作被实际标记为成功之前，它们在所有的成员上都已经成功了。我们稍后将探讨写关注。

8. 可行的复制部署

你选择用来部署一个副本集的架构会影响其功能和规模。在本节中，将介绍一些策略，在决定采用何种架构时，需要知道这些策略。我们也会探讨部署架构。

(1) 奇数个成员：在选举一个主节点时，应该使用奇数个成员以便确保不会出现票数相等的情况。如果节点数量是偶数，那么可以使用一个仲裁者来确保参与选举的节点总数是奇数，如图 7-6 所示。

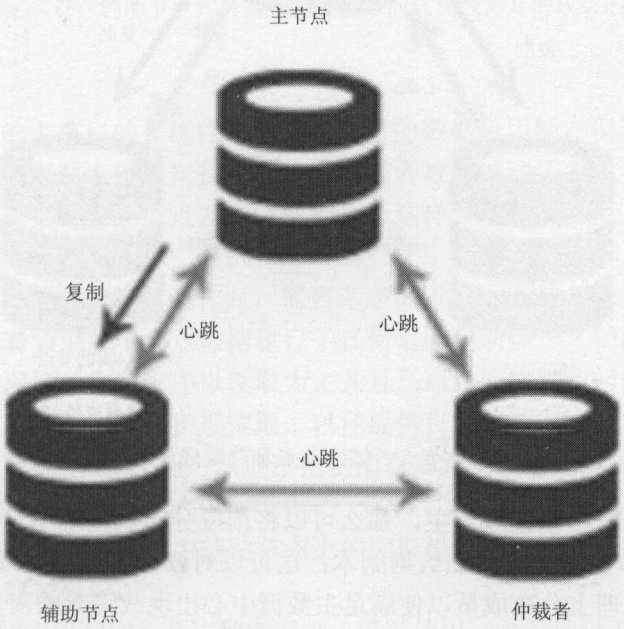

图 7-6　带有主节点、辅助节点和仲裁者的成员副本集

(2) 副本集容错功能是允许关闭的成员数量，这些成员可以关闭但是在出现故障时副本集仍旧具有足够的成员来选举一个主节点。表 7-1 表明了副本集中的成员数量与其容错功能之间的关系。在决定成员数量时就应该考虑容错功能。

表 7-1 副本集容错功能

成 员 数 量	选举一个主节点所需要的多数数量	容 错 功 能
3	2	1
4	3	1
5	3	2
6	4	2

(3) 如果应用程序具有特定的专用需求，比如报告或备份，那么延迟或隐藏成员可以被考虑作为副本集的一部分，如图 7-7 所示。

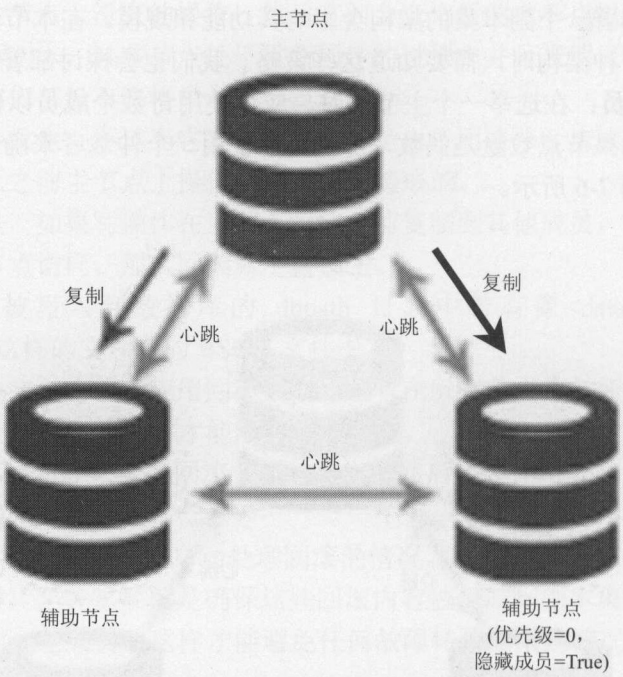

图 7-7 带有主节点、辅助节点和隐藏成员的成员副本集

(4) 如果应用程序以读取为主，那么可以将读取分布到辅助节点上。随着需求的增加，可以添加更多的节点来增加数据副本；这可以对读取的吞吐量产生积极影响。

(5) 应该在地理上分布成员以便满足主数据中心出现故障时的需要。如图 7-8 所示，可以将存留在一个地理上与主数据中心不同的位置的成员的优先级设置为 0，这样它们就不能被选举为主节点并且仅可以充当备份。

(6) 当副本集成员跨数据中心分布时，网络分区会阻止这些数据中心彼此通信。为了确保在网络分区的情况下能够得到多数票，将在同一位置中存留多数成员。

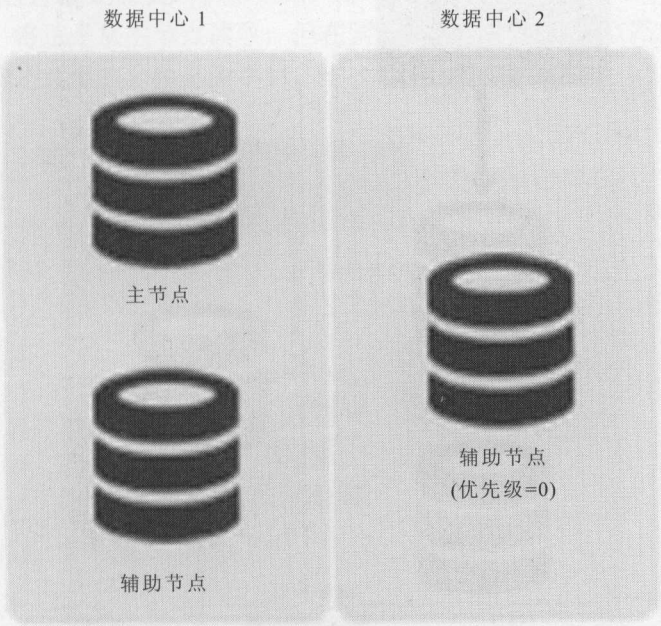

图 7-8 带有主节点、辅助节点和跨数据中心分布的优先级为 0 的成员的成员副本集

9. 扩展读取

尽管辅助节点的主要目的是确保主节点关闭时的数据可用性，但辅助节点还可用作其他合理用途。它们可被专门用来执行备份操作或者数据处理任务或者扩展读取。扩展读取的其中一种方式是对辅助节点发出读取查询；这样一来，主节点上的工作负荷就会降低。

在将辅助节点用于扩展读取操作时，需要考虑的重要一点是，在 MongoDB 中复制是异步的，这意味着如果对主节点数据执行了任何写或更新操作，辅助副本的数据将短暂过时。如果正在探讨的应用程序以读取为主并且是通过网络来访问的，也无需最新的数据，那么辅助节点就可以用于扩展读取，以便提供良好的读取吞吐量。尽管默认情况下读取请求会被路由到主节点，但可以通过指定读取偏好来将这些请求分布到辅助节点。图 7-9 描述了默认的读取偏好。

下面是理想的用例，凭此路由辅助节点上的读取操作可以帮助获得读取吞吐量的显著提升，也有助于降低延迟：

(1) 地理分布的应用程序：在这种情况下，可以使用一个跨地理位置分布的副本集。应该将读取偏好设置为从最近的辅助节点读取。这有助于降低由跨网络读取造成的延迟，并且会提升读取的性能。参见图 7-10。

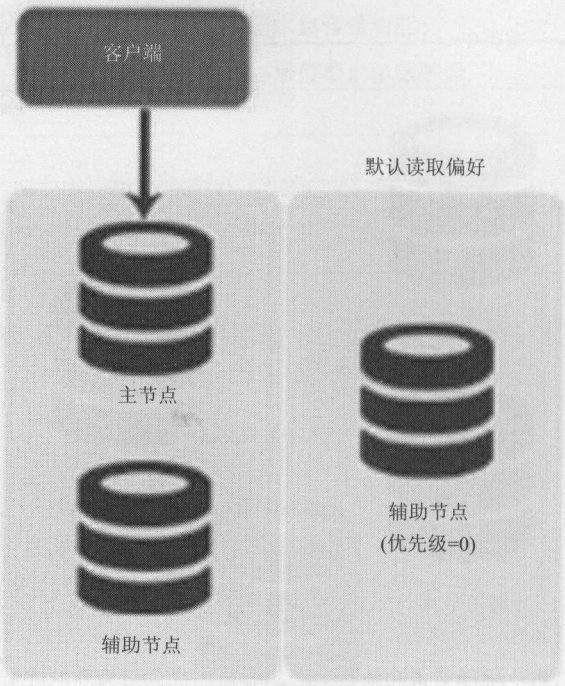

图 7-9　默认读取偏好

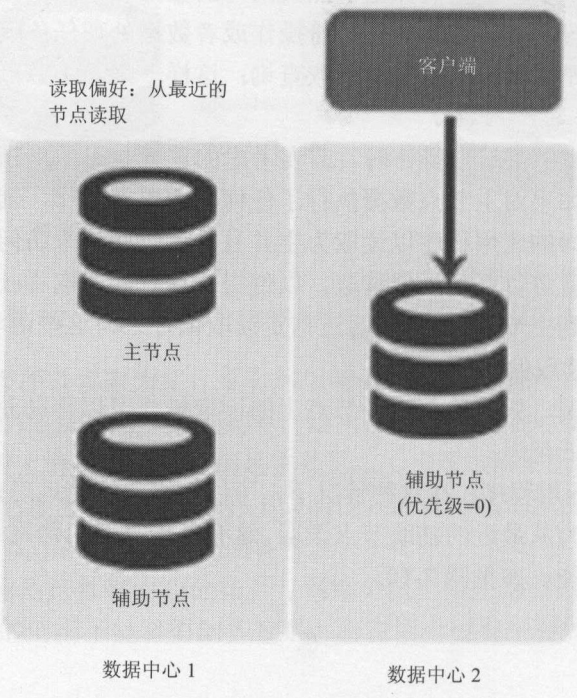

图 7-10　读取偏好——从最近的节点读取

(2) 如果应用程序总是需要最新的数据，那么它会使用选项 primaryPreferred，这样在常规的环境中，将总是从主节点读取，但在紧急情况下，会将读取路由到辅助节点。这在故障转移期间是有用的。参见图 7-11。

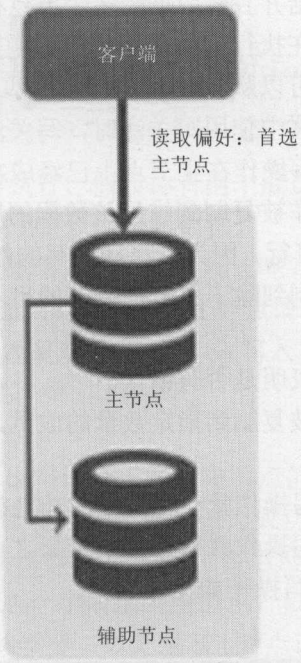

图 7-11 读取偏好——首选主节点

(3) 如果有一个支持两类操作的应用程序，第一个操作是涉及读取和对数据进行一些处理的主要工作负荷，而第二个操作会生成使用数据的报告。在这种情况下，可以将报告读取指向辅助节点。

MongoDB 支持以下读取偏好模式：

- **primary**：这是默认模式。所有的读取请求都会被路由到主节点。
- **primaryPreferred**：在常规环境中，会从主节点读取，但在像主节点不可用这样的紧急情况下，将从辅助节点读取。
- **secondary**：从辅助成员读取。
- **secondaryPreferred**：从辅助成员读取。如果辅助节点不可用，则从主节点读取。
- **nearest**：从最近的副本集成员读取。

除了扩展读取之外，使用辅助节点的第二个理想用例是转移密集的处理、聚合以及管理任务，以避免主节点性能的降低。可以在辅助节点上执行阻塞操作，而永远不会影响主节点的性能。

10. 应用程序写关注

当客户端应用程序与 MongoDB 交互时，它通常不会知道数据库是独立部署还是作为副本集来部署的。不过，在处理副本集时，客户端应该要清楚写关注和读关注。

由于一个副本集会复制数据并且将数据跨多个节点存储，因此这两个关注为客户端应用程序提供了灵活性，以便在执行读或写操作时强制数据跨节点保持一致。

使用写关注使得应用程序可以从 MongoDB 得到成功或失败的响应。

在 MongoDB 的副本集部署中使用写关注时，写关注会从服务器将一条确认消息发送到应用程序，该消息会表明写操作在主节点上已经成功了。不过，可以就这一点进行配置，以便写关注仅在该写操作被复制到维护该数据的所有节点时才返回成功。

在实际的场景中，这并不可行，因为它将降低写的性能。理论上，客户端可以确保，在使用写关注时，数据会被复制到除了主节点之外的另一个节点，这样一来，即使主节点关闭，该数据也不会丢失。

写关注会返回一个对象，表明是否有错误。

w 选项会确保写操作已经被复制到指定数量的成员。若干个或多数个成员都可以被指定为 w 选项的值。

如果指定了一个数字，则写操作就会在返回成功之前复制到该指定数量的节点。如果指定了多数个这一数量，则写操作就会在返回结果之前复制到过半数的成员。

图 7-12 显示了使用 w:2 时写操作如何发生。

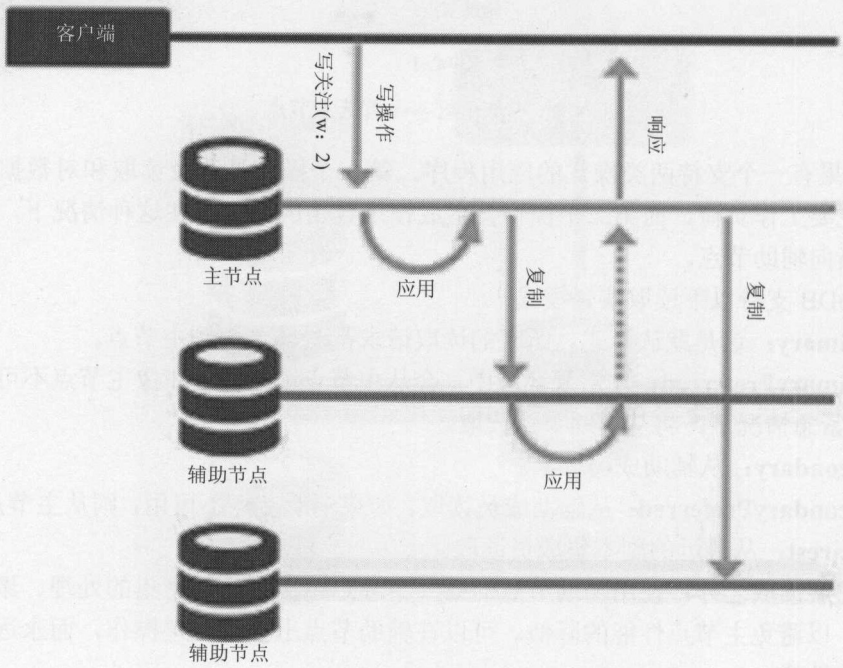

图 7-12 写关注

如果在指定数量时，该数量大于实际持有数据的节点数量，则该命令会持续等待直到成员可用。为了避免这一不确定的等待时长，还应该将超时设置与 w 一起使用，这样就能确保仅等待指定的时长，并且如果到时写操作还没有成功，就会超时。

使用写关注时写操作如何发生

比如，为了确保写入的数据呈现在至少两个成员上，需要运行以下命令：

```
>db.testprod.insert({i:"test", q: 50, t: "B"}, {writeConcern: {w:2}})
```

为了理解这个命令将被如何执行，假设你有两个成员，一个被指定为主节点，另一个被指定为辅助节点，并且它从主节点同步其数据。

但是主节点如何才会知道辅助节点同步的时间点呢？由于为了应用 op 结果，辅助节点会查询主节点的 oplog，所以如果辅助节点在 t 时点请求一个 op 写入，那么就会向主节点表明，在 t 之前辅助节点已经复制了所有的 op 写入。

下面是写关注要进行的步骤。

(1) 写操作会被导向主节点。

(2) 会使用描述操作时长的 ts 将该操作写到主节点的 oplog。

(3) 执行了 w: 2，这样写操作就需要在被标记为成功之前被写到另一台服务器。

(4) 辅助节点会为该 op 查询主节点的 oplog，并且应用该 op。

(5) 接下来，辅助节点会发送一个请求到主节点，以查询 ts 大于 t 的 op。

(6) 此时，主节点会发送一个更新，以记录在 t 时点该操作已经被辅助节点应用，因为辅助节点使用了{ts: {$gt: t}}请求 op。

(7) 写关注发现，在主节点和辅助节点上都发生了一个写操作，满足 w: 2 的条件，进而该命令会返回成功。

7.4.3 实现带有副本集的高级群集

在学习了副本集的架构及其内部运行原理之后，你现在要专注于副本集的管理和使用。你要专注于以下方面：

(1) 设置一个副本集。

(2) 移除一台服务器。

(3) 添加一台服务器。

(4) 添加一个仲裁者。

(5) 检查状态。

(6) 强制进行新的主节点选举。

(7) 使用网络接口来检查副本集的状态。

以下示例假设了一个名称为 testset 的副本集，它具有如表 7-2 所示的配置。

表 7-2 副本集配置

成员	守护程序	主机:端口	数据文件路径
Active_Member_1	Mongod	[hostname]:27021	C:\db1\active1\data
Active_Member_2	Mongod	[hostname]:27022	C:\db1\active2\data
Passive_Member_1	Mongod	[hostname]:27023	C:\db1\passive1\data

可以使用以下命令来找出表 7-2 中使用的 hostname:

```
C:\>hostname
ANOC9
C:\>
```

在以下示例中，需要在你的系统上使用 hostname 命令返回的值来替换[hostname]。在我们的例子中，所返回的值是 ANOC9，它会用于后面的示例中。

在后面的实现中使用默认的(MMAPv1)存储引擎。

1. 设置一个副本集

为了设置副本集并且让它运行起来，需要让所有的活跃成员启动并且运行。

第一步是启动第一个活跃成员。打开一个终端窗口并且创建数据目录：

```
C:\>mkdir C:\db1\active1\data
C:\>
```

连接到该 mongod：

```
    c:\practicalmongodb\bin>mongod --dbpath C:\db1\active1\data --port 27021
--replSet
    testset/ANOC9:27021 -rest

    2015-07-13T23:48:40.543-0700 I CONTROL ** WARNING: --rest is specified
without --httpinterface,
    2015-07-13T23:48:40.543-0700 I CONTROL ** enabling http interface
    2015-07-13T23:48:40.543-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero-out data files
    2015-07-13T23:48:40.563-0700 I JOURNAL [initandlisten] journal
dir=C:\db1\active1\data\journal
    2015-07-13T23:48:40.564-0700 I JOURNAL [initandlisten] recover : no
journal files present,no recovery needed
    ..................................... port=27021 dbpath=C:\db1\active1\
data 64-bit
                                          host=ANOC9
    2015-07-13T23:48:40.614-0700 I CONTROL [initandlisten] targetMinOS:
Windows 7/Windows Server 2008 R2
    2015-07-13T23:48:40.615-0700 I CONTROL [initandlisten] db version v3.0.4
```

如你所见，–replSet 选项会指定实例联结到的副本集的名称以及该副本集的一个成员的名称，在上面的示例中就是 Active_Member_2。

尽管上面的示例中仅指定了一个成员，但可以通过指定逗号分隔的地址来提供多个成员，就像这样：

```
mongod -dbpath C:\db1\active1\data -port 27021 -replset
testset/[hostname]:27022,[hostname]:27023 --rest
```

在接下来的步骤中，你要让第二个活跃成员启动和运行。在新的终端窗口中为第二个活跃成员创建数据目录。

```
C:\>mkdir C:\db1\active2\data
C:\>
```

连接到 mongod：

```
c:\ practicalmongodb \bin>mongod --dbpath C:\db1\active2\data --port
27022 -replSet testset/ANOC9:27021 -rest
2015-07-13T00:39:11.599-0700 I CONTROL ** WARNING: --rest is specified
without --httpinterface,
2015-07-13T00:39:11.599-0700 I CONTROL ** enabling http interface
2015-07-13T00:39:11.604-0700 I CONTROL Hotfix KB2731284 or later update
is installed, noneed to zero-out data files
2015-07-13T00:39:11.615-0700 I JOURNAL [initandlisten] journal
dir=C:\db1\active2\data\journal
2015-07-13T00:39:11.615-0700 I JOURNAL [initandlisten] recover : no
journal files present,no recovery needed
2015-07-13T00:39:11.664-0700 I JOURNAL [durability] Durability thread
started
2015-07-13T00:39:11.664-0700 I JOURNAL [journal writer] Journal writer
thread startedrs.initiate() in the shell -- if that is not already done
```

最后，需要启动被动成员。打开一个单独的窗口并且为该被动成员创建数据目录。

```
C:\>mkdir C:\db1\passive1\data
C:\>
```

连接到 mongod：

```
c:\ practicalmongodb \bin>mongod --dbpath C:\db1\passive1\data --port
27023 -replSet testset/ ANOC9:27021 -rest
2015-07-13T05:11:43.746-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero-out data files
2015-07-13T05:11:43.757-0700 I JOURNAL [initandlisten] journal
dir=C:\db1\passive1\data\journal
2015-07-13T05:11:43.808-0700 I CONTROL [initandlisten] MongoDB starting :
pid=620 port=27019 dbpath=C:\db1\passive1\data 64-bit host= ANOC9
```

```
.....................................................................
    2015-07-13T05:11:43.812-0700 I CONTROL [initandlisten] options: { net:
{ http:{ RESTInterfaceEnabled: true, enabled: true }, port: 27019 },
replication: { re lSet: "testset/ ANOC9:27017" }, storage: { dbPath:
"C:\db1\passive1\data" }
```

在前面的示例中，--rest 选项被用于激活端口+1000 上的一个 REST 接口。激活 REST 使得可以使用网络接口检查副本集状态。

在上述步骤结束时，你就有了 3 台服务器处于启动和运行状态，并且彼此能够通信；不过该副本集仍旧没有初始化。在接下来的步骤中，你要初始化该副本集并且指示每个成员的职责和角色。

为了初始化副本集，你要连接到其中一台服务器。在这个示例中，这台服务器就是第一台服务器，它运行在端口 27021 上。

打开一个新的命令提示符，并且连接到第一台服务器的 mongo 接口：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo ANOC9 --port 27021
MongoDB shell version: 3.0.4
connecting to: ANOC9:27021/test
>
```

接下来，切换到 admin 数据库：

```
> use admin
switched to db admin
>
```

接下来，设置一个配置数据结构，它提供了与服务器有关的角色：

```
>cfg = {
... _id: 'testset',
... members: [
... {_id:0, host: 'ANOC9:27021'},
... {_id:1, host: 'ANOC9:27022'},
... {_id:2, host: 'ANOC9:27023', priority:0}
... ]
... }
{        "_id" : "testset",
        "members" : [
                {
                        "_id" : 0,
                        "host" : "ANOC9:27021"
                },
..........
                {
                        "_id" : 2,
```

```
                "host" : "ANOC9:27023",
                "priority" : 0
            } ]}>
```

遵循此步骤，就能配置副本集结构。

在定义被动成员的角色时你已经使用了 0 优先级。这意味着该成员无法被提升为主节点。

下面的命令会初始化副本集：

```
>rs.initiate(cfg)
{ "ok" : 1}
```

我们现在检查副本集的状态以便检查它是否被正确设置：

```
testset:PRIMARY> rs.status()
{
   "set" : "testset",
"date" : ISODate("2015-07-13T04:32:46.222Z")
   "myState" : 1,
"members" : [
      {
            "_id" : 0,
.........................
testset:PRIMARY>
```

该输出结果表明一切都 OK。该副本集现在已经被成功配置并且初始化了。

我们来看看如何才能判定主节点。为此，你要连接到任意一个成员并且运行以下命令来验证主节点：

```
testset:PRIMARY> db.isMaster()
{
        "setName" : "testset",
        "setVersion" : 1,
        "ismaster" : true,
        "primary" : " ANOC9:27021",
        "me" : "ANOC9:27021",
        ........................................
        "localTime" : ISODate("2015-07-13T04:36:52.365Z"),
        ..................................................
        "ok" : 1
}testset:PRIMARY>
```

2. 移除一台服务器

在这个示例中，要从副本集中移除第二个活跃成员。我们连接到第二个成员的 mongo 实例。打开一个新的命令提示符，如以下代码所示：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo ANOC9 --port 27022
MongoDB shell version: 3.0.4
connecting to: 127.0.0.1:27022/ANOC9
testset:SECONDARY>

Issue the following command to shut down the instance:
testset:SECONDARY> use admin
switched to db admin
testset:SECONDARY> db.shutdownServer()
2015-07-13T21:48:59.009-0700 I NETWORK DBClientCursor::init call()
failed server should be down...
```

接着，需要连接到主成员 mongo 控制台并且执行以下命令来移除该成员：

```
testset:PRIMARY> use admin
switched to db admin
testset:PRIMARY> rs.remove("ANOC9:27022")
{ "ok" : 1 }
testset:PRIMARY>
```

为了验证该成员是否被移除，可以运行 rs.status()命令。

3. 添加一台服务器

接着你要将一个新的活跃成员添加到副本集。就像处理其他成员一样，首先要打开一个新的命令提示符并且先要创建数据目录：

```
C:\>mkdir C:\db1\active3\data
C:\>
```

接下来，要使用以下命令启动 mongod：

```
c:\practicalmongodb\bin>mongod --dbpath C:\db1\active3\data --port 27024
--replSet testset/
ANOC9:27021 --rest
..........
```

你已经让新的 mongod 运行了，所以现在需要将这个成员添加到副本集。为此要连接到主节点的 mongo 控制台：

```
C:\>c:\practicalmongodb\bin\mongo.exe --port 27021
MongoDB shell version: 3.0.4
connecting to: 127.0.0.1:27021/test
testset:PRIMARY>
```

接下来，要切换到 admin db：

```
testset:PRIMARY> use admin
```

```
switched to db admin
testset:PRIMARY>
```

最后，需要运行以下命令来将新的 mongod 添加到副本集：

```
testset:PRIMARY> rs.add("ANOC9:27024")
{ "ok" : 1 }
```

可以使用 rs.status()通过检查该副本集的状态来验证这个新的活跃成员是否被添加。

4. 将一个仲裁者添加到副本集

在这个示例中，要将一个仲裁者成员添加到副本集。就像处理其他成员一样，首先要为 MongoDB 实例创建数据目录：

```
C:\>mkdir c:\db1\arbiter\data
C:\>
```

接下来要使用以下命令启动 mongod：

```
c:\practicalmongodb\bin>mongod --dbpath c:\db1\arbiter\data --port 30000
--replSet testset/
ANOC9:27021 --rest
2015-07-13T22:05:10.205-0700 I CONTROL [initandlisten] MongoDB starting :
pid=3700
port=30000 dbpath=c:\db1\arbiter\data 64-bit host=ANOC9
.....................................................
```

连接到主节点的 mongo 控制台，切换到 admin db，并且将新创建的 mongod 作为仲裁者添加到副本集：

```
C:\>c:\practicalmongodb\bin\mongo.exe --port 27021
MongoDB shell version: 3.0.4
connecting to: 127.0.0.1:27021/test
testset:PRIMARY> use admin
switched to db admin

testset:PRIMARY> rs.addArb("ANOC9:30000")
{ "ok" : 1 }
testset:PRIMARY>
```

可以使用 rs.status()来验证这个步骤是否成功。

5. 使用 rs.status()检查状态

在上面所有的示例中我们一直在使用 rs.status()来检查副本集状态。在本节中，你要学习这个命令到底是什么。

它使你可以检查所连接到的控制台所属成员的状态，也可以让你查看这些成员在副

本集中的角色。

以下命令是在主节点的 mongo 控制台中运行的：

```
testset:PRIMARY> rs.status()
{
     "set" : "testset",
"date" : ISODate("2015-07-13T22:15:46.222Z")
     "myState" : 1,
"members" : [
     {
               "_id" : 0,
.........................
     "ok" : 1
testset:PRIMARY>
```

myState 字段的值表明了该成员的状态，并且它可以具有如表 7-3 所示的值。

表 7-3　副本集状态

myState	描　述
0	阶段 1，启动
1	主节点成员
2	辅助节点成员
3	恢复状态
4	致命错误状态
5	阶段 2，启动
6	未知状态
7	仲裁者成员
8	关闭或无法连接
9	当一个写操作从主节点迁移成辅助节点之后被该节点回滚时，就会变成这个状态
10	从副本集移除成员时，该成员就会进入这个状态

因此，上述命令会将 myState 的值返回为 1，它表明这是一个主节点成员。

6. 强制进行一次新选举

可以使用 rs.stepDown()命令强制当前主服务器关闭。这一强制行为会启动新主节点的选举。

这个命令在以下场景中很有用：

(1) 当在模拟主节点故障的影响时，可以强制群集进行故障转移。这样你就可以测试在这样的场景中你的应用程序会如何响应。

(2) 当主服务器需要离线时。这样做要么是为了进行维护，要么是为了升级或者检

查该服务器。

(3) 当需要针对数据结构运行诊断程序时。

下面是在针对 testset 副本集运行该命令时的输出结果：

```
testset:PRIMARY> rs.stepDown()
2015-07-13T22:52:32.000-0700 I NETWORK DBClientCursor::init call() failed
2015-07-13T22:52:32.005-0700 E QUERY Error: error doing query: failed
2015-07-13T22:52:32.009-0700 I NETWORK trying reconnect to
127.0.0.1:27021 (127.0.0.1) failed
2015-07-13T22:52:32.011-0700 I NETWORK reconnect 127.0.0.1:27021
(127.0.0.1) ok
                                                    testset:SECONDARY>
```

在该命令执行后，提示符会从 testset:PRIMARY 变成 testset:SECONDARY。

可以使用 rs.status()来检查 stepDown()是否执行成功。

请注意，它返回的 myState 的值现在是 2，这意味着“成员正作为辅助节点运行”。

7. 使用网络接口检查副本集的状态

MongoDB 保留有一个基于网络的控制台，以用于浏览系统状态。在你的示例中，可以通过 http://localhost:28021 来访问这个控制台。

默认情况下，网络接口端口号会被设置为 X+1000，其中 X 是 mongod 实例端口号。在本章的示例中，由于主节点实例位于 27021 上，因此该网络接口位于端口 28021 上。

图 7-13 显示了指向副本集状态的一个链接。单击该链接会向你显示如图 7-14 所示的副本集仪表盘。

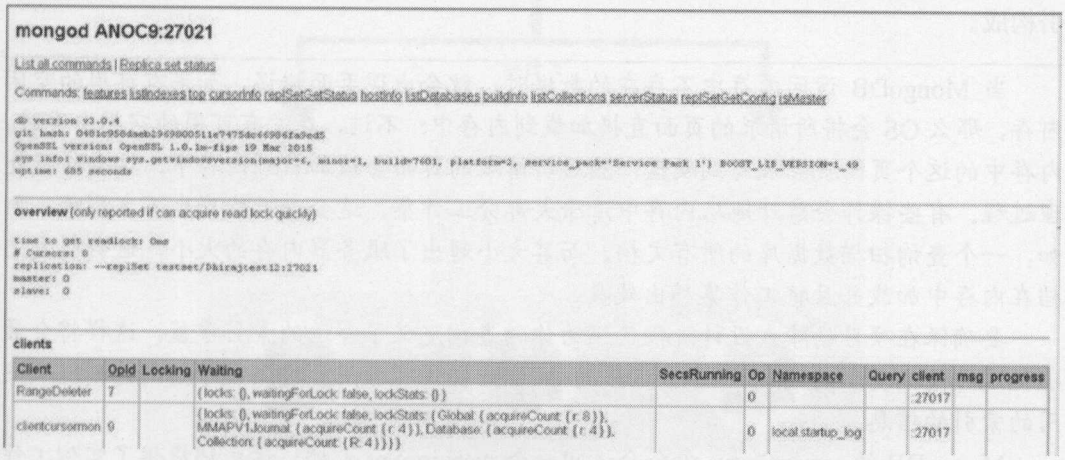

图 7-13 网络接口

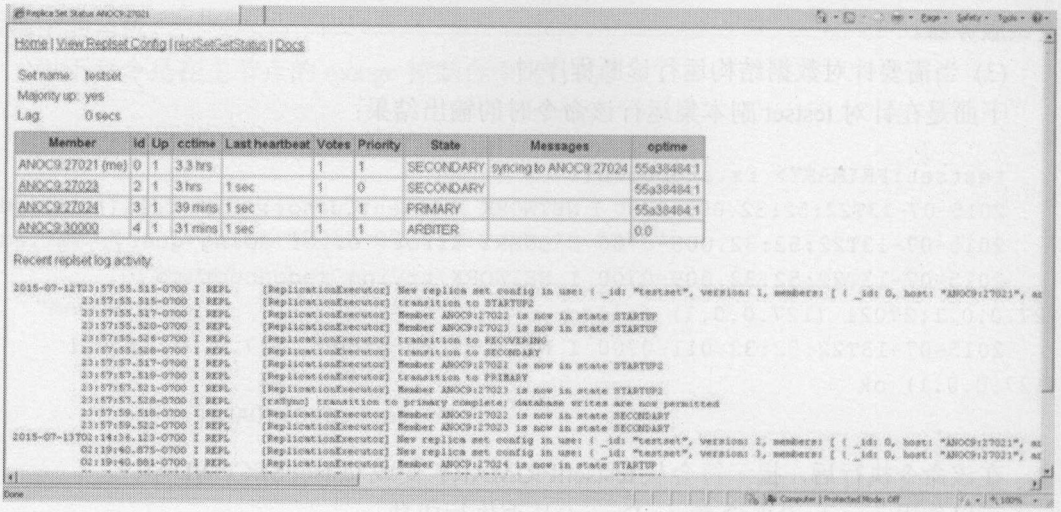

图 7-14　副本集状态报告

7.5　分片

在上一节中，你看到了如何在 MongoDB 中使用副本集复制数据以便防止出现任何问题以及如何用副本集来分布读取负荷以便提高读取效率。

MongoDB 大量地将内存用于低延迟的数据库操作。当对比从内存读取数据以及从硬盘读取数据的速度时就会发现，从内存读取要比从硬盘读取快大约 100 000 倍。

在 MongoDB 中，理论上工作集应该适合于内存。工作集由最频繁访问的数据和索引构成。

当 MongoDB 访问内存中不存在的数据时，就会出现页面错误。如果有可用的空闲内存，那么 OS 会将所请求的页面直接加载到内存中；不过，在没有可用的空闲内存时，内存中的这个页面就会被写到硬盘，然后所请求的页面会被加载到内存中，这会降低处理过程。有些操作会意外地从内存中清除大部分工作集，这会对性能产生不良影响。例如，一个查询扫描数据库的所有文档，而其大小超出了服务器内存的大小，这会引发文档在内存中加载并且将工作集移出硬盘。

要确保在项目的模式设计阶段已经为你的查询定义了合适的索引覆盖，这样将会最小化这种情况发生的风险。MongoDB 的 explain 操作可被用于提供关于查询计划以及使用的索引的信息。

MongoDB 的 serverStatus 命令会返回一个 workingSet 文档，该文档提供了实例工作集大小的一个估算值。运营团队可以跟踪在一段指定时间段内实例访问了多少个页面以及工作集最老和最新的文档之间的时间差。跟踪所有这些指标，就可以查明工作集将在何时达到当前的内存限制，这样一来就可以采取积极措施来确保系统扩展得足够大以便应对这一情况。

在 MongoDB 中，扩展是通过横向扩展数据来处理的(比如，跨多个廉价服务器对数据分区)，这也称为分片(横向扩展)。

通过跨服务器横向划分数据集，其中每台服务器都负责处理自己的数据部分并且没有一台服务器会负荷过重，这样分片就可以应对扩展的挑战以支持大数据集和高吞吐量。这些服务器也被称为分片服务器。

每一个分片都是一个独立的数据库。所有这些分片共同组成了单个逻辑数据库。

分片会减少每个分片处理的操作数量。例如，在插入数据时，只需要访问负责存储那些记录的分片就可以了。

需要由每个分片处理的程序会随着群集的增长而减少，这是因为分片持有的数据子集缩小了。这会引发吞吐量和规模的横向增长。

我们假设你有一个大小为 1TB 的数据库。如果分片的数量是 4，那么每个分片大约要处理 265GB 的数据，而如果分片的数量增加到 40，那么每个分片将仅持有 25GB 的数据。

图 7-15 描述了在跨 3 个分片分布时，一个被分片的集合看起来会是什么样。

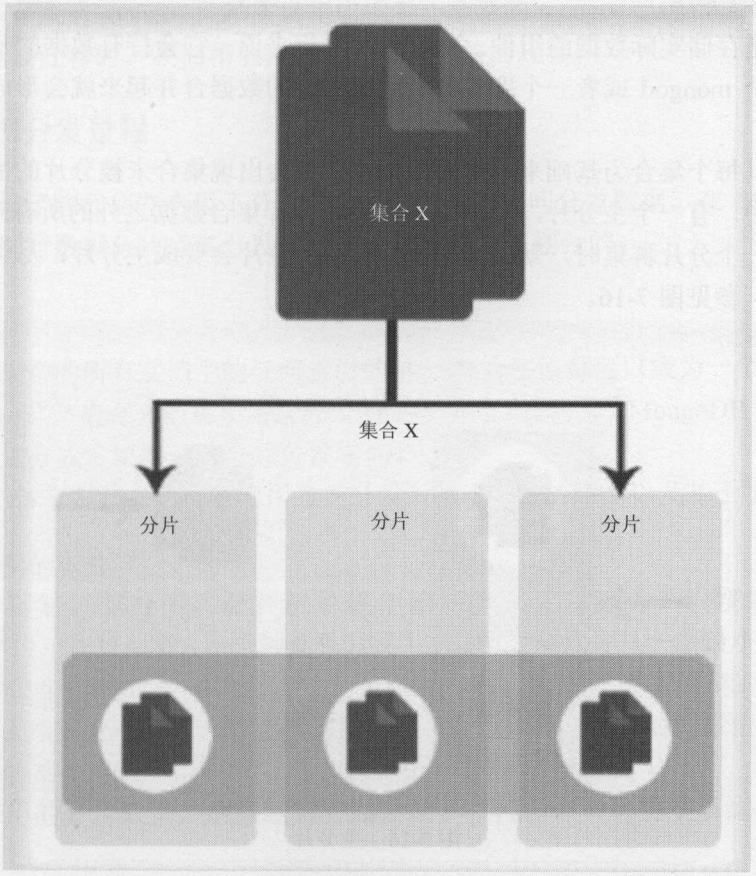

图 7-15　跨 3 个分片的分片集合

尽管分片是一个吸引人且强大的功能，但是它对于基础设施的需求也很高，并且它

会增加整体部署的复杂性。因此需要理解你可能会考虑使用分片的应用场景。

可以在以下实例中使用分片：

- 数据集的大小很大并且它已经开始挑战单一系统的容量。
- 由于 MongoDB 使用内存来快速抓取数据，因此在预计将达到活跃工作集限制时，扩展就变得很重要了。
- 如果应用程序以写入为主，则可以使用分片来跨多台服务器分散写操作。

7.5.1 分片组件

接下来将了解在 MongoDB 中启用分片的组件。在 MongoDB 中是通过分片群集来启用分片的。

下面就是一个分片群集的组件：

- 分片
- mongos
- 配置服务器

分片就是存储实际数据的组件。对于分片群集来说，它会持有数据的一个子集并且它可以是一个 mongod 或者一个副本集。所有分片的数据合并起来就会形成分片群集的完整数据集。

分片是以每个集合为基础来启用的，因此可能会出现集合未被分片的情况。在每一个分片群集中，有一个主分片，其中会放置除了分片集合数据之外的所有未分片集合。

在部署一个分片群集时，默认情况下，第一个分片会变成主分片，尽管这个主分片是可配置的。参见图 7-16。

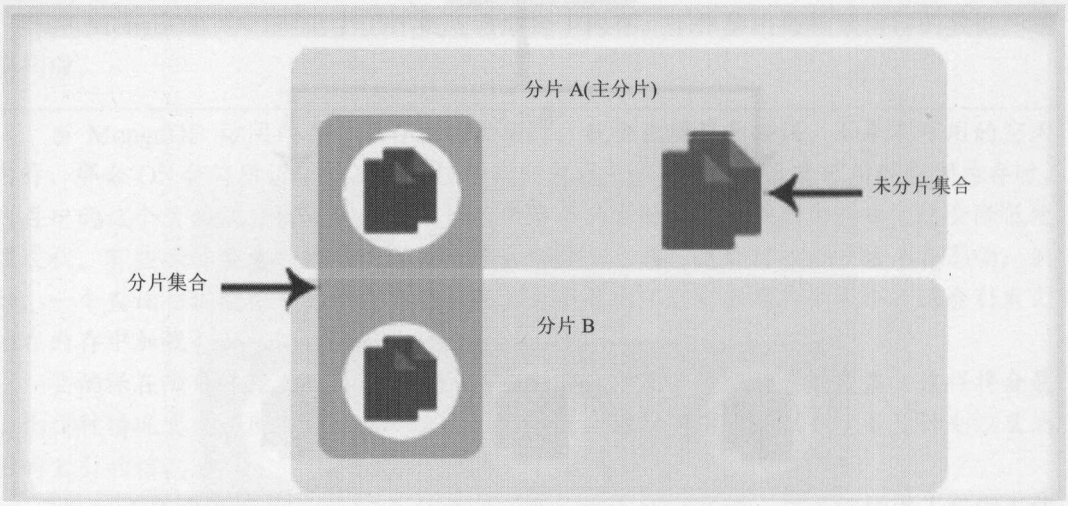

图 7-16 主分片

配置服务器是特殊的 mongod，它们会持有分片群集的元数据。这些元数据描述了分片系统的状态和组织。

配置服务器会存储用于单个分片的群集的数据。为了群集能够正常运行，配置服务

器应该可供使用。

一台配置服务器可能会造成群集的单点故障。对于生产环境部署，建议至少使用 3 台配置服务器，这样一来，即使一台配置服务器不可访问，群集也可以保持正常运行。

配置服务器会将数据存储在配置数据库中，这使得我们可以将不同的客户端请求路由到各自的数据。这个数据库不应该被更新。

只有在出于平衡群集的原因对数据分布变更时，MongoDB 才会将数据写到配置服务器。

mongos 充当了路由器。它们负责将读取和写入请求从应用程序路由到分片。

与一个 mongo 数据库交互的应用程序不需要关心在分片内部是如何存储数据的。对于它们来说，这是透明的，因为它们仅与 mongos 交互。接着，mongos 会将读取和写入路由到分片。

mongos 会缓存来自配置服务器的元数据，这样一来，对于每一个读取和写入请求，它们就不会让配置服务器承担过重的负荷。

不过，在以下情况中，是从配置服务器读取数据的：

- 在一个现有 mongos 已经重启完成或者一个新的 mongos 已经首次启动时。
- 数据块迁移时。我们稍后将详尽阐释数据块的迁移。

7.5.2 数据分发过程

接下来将了解如何在启用了分片的集合中的分片之间分发数据。在 MongoDB 中，会在集合级别对数据分片或者分发。集合是由分片键来划分的。

分片键

存在于集合的所有文档中的任何索引的单一/复合字段都可以成为一个分片键。你要指定分片键，这是集合文档需要用来分发的字段基础。在内部，MongoDB 会根据该字段的值将文档划分成多块并且跨分片分发它们。

MongoDB 有两种方式可以启用数据分发：基于范围的划分以及基于哈希值的划分。

基于范围的划分

在基于区域的划分中，分片键的值会被划分成多个范围。假设你考虑将一个 timestamp 字段用作分片键。在这种划分方式中，其值会被视作从一条最小值到最大值的直线，其中最小值是起始时间点(比如，01/01/1970)，而最大值是结束时间点(比如，12/31/9999)。该集合中的每一个文档都只会具有在这一范围内的时间戳值，并且它将代表该条线上的某个时点。

基于可用的分片数量，这条线将被划分成多个范围，并且将基于这些范围来分发文档。

在这种划分模式中，如图 7-17 所示，接近该分片键值的文档很可能会落在相同的分片上。这会显著提升范围查询的性能。

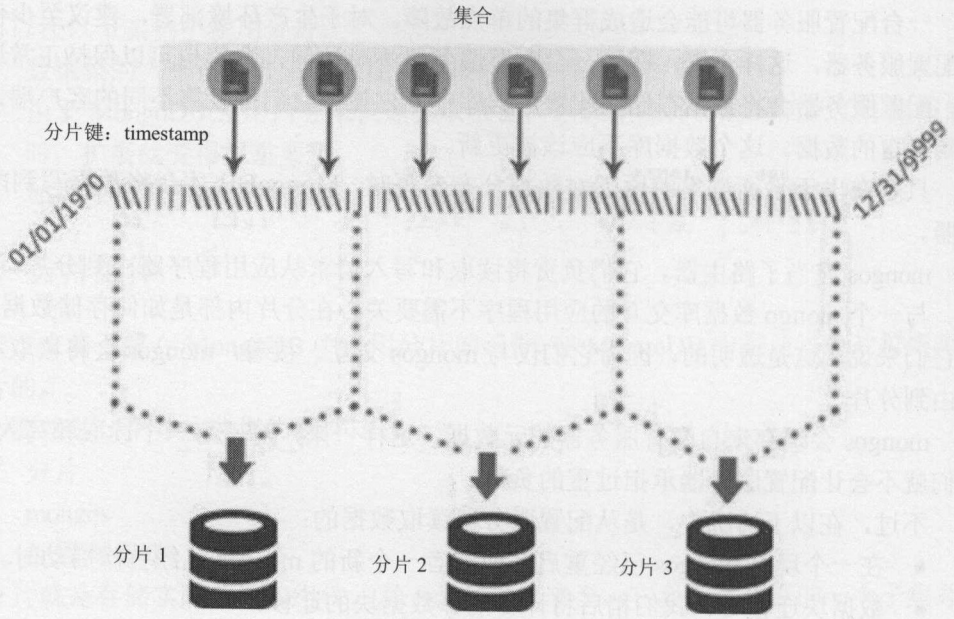

图 7-17 基于范围的划分

不过，其缺点是，它会导致数据的不均匀分布，从而导致其中一个分片负荷过重，该分片可能最终会接收到大多数请求，而其他分片仍旧负荷不足，因此系统就没有正确扩展。

基于哈希值的划分

在基于哈希值的划分中，数据是基于分片字段的哈希值来分发的。如果选择了它，那么相较于基于范围的划分来说，就会造成更为随机的分布。

分片键接近的文档也不太可能是相同数据块的一部分。例如，对于基于_id 字段哈希值的范围来说，会存在一条哈希值的直线，它将再次基于分片的数量来划分。在哈希值的基础上，文档将位于任何一个分片中。参见图 7-18。

与基于范围的划分相反，这会确保数据均匀分布，但它的应用会降低范围查询的效率。

数据块

数据会以数据块的形式在分片之间移动。分片键范围会被进一步划分成子范围，这也被称为数据块。参见图 7-19。

对于一个分片群集来说，64MB 是默认的数据块大小。在大多数情况下，这对于数据块分割和迁移来说是合适的大小。

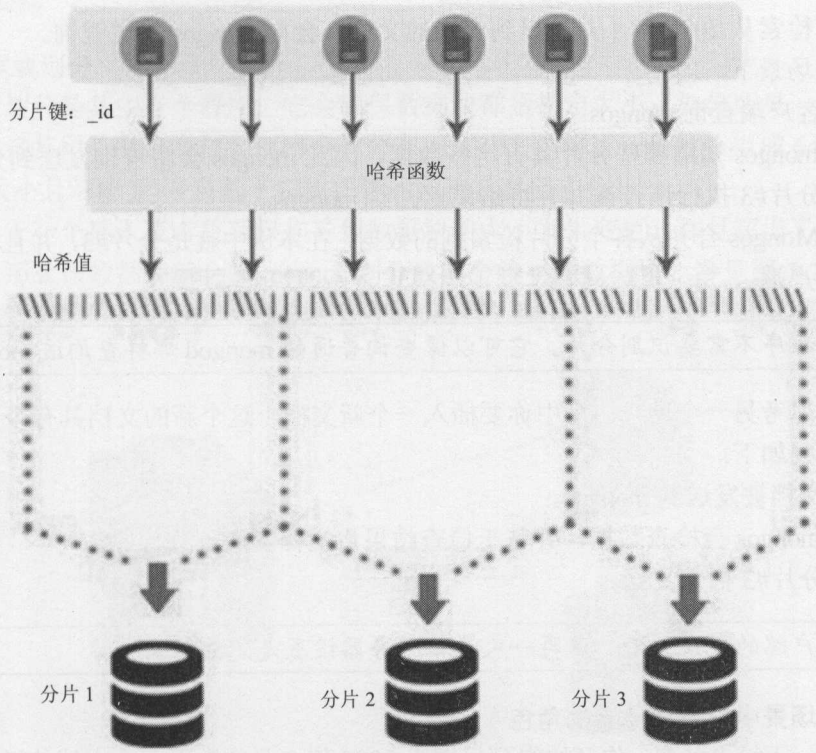

图 7-18 基于哈希值的划分

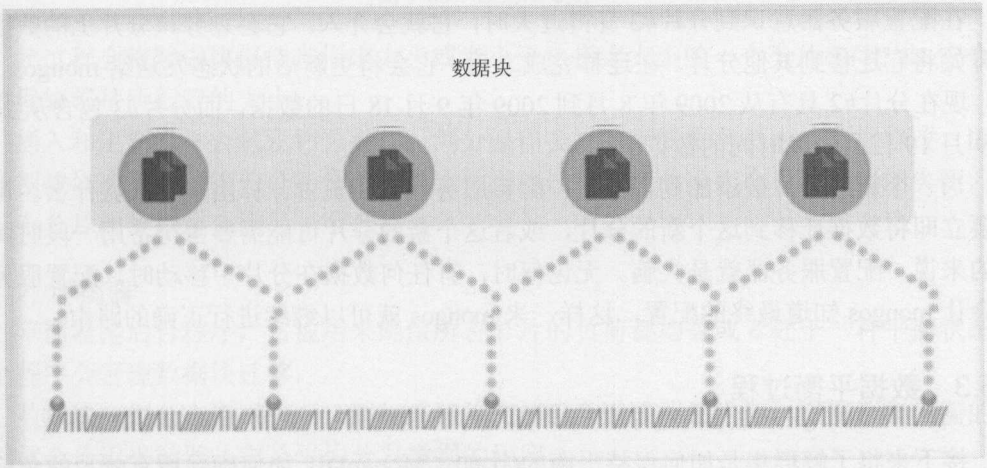

图 7-19 数据块

我们使用一个示例探讨分片和数据块的执行。假设有一个博文帖子集合，它是基于字段 date 来分片的。这表明将基于 date 字段值来划分该集合。我们进一步假设有 3 个分片。在这样的情况下，数据可能会跨分片分布，如下所示：

分片#1：从最早的时间到 2009 年 7 月

分片#2：从 2009 年 8 月到 2009 年 12 月

分片#3：从 2010 年 1 月到最晚的时间

为了检索从 2010 年 1 月 1 日到今天的文档，会向 mongos 发送查询。

在该场景下，

(1) 客户端查询 mongos。

(2) mongos 知道哪些分片具有这些数据，因此 mongos 会将查询发送到分片#3。

(3) 分片#3 执行该查询并且将结果返回给 mongos。

(4) Mongos 合并从各个分片检索到的数据，在本例中就是分片#3，并且将最终结果返回给客户端。

应用程序不需要识别分片。它可以像查询普通的 mongod 那样查询该 mongos。

我们思考另一个场景，其中你要插入一个新文档。这个新的文档具有今天的日期。事件的序列如下：

(1) 文档被发送到 mongos。

(2) mongos 会检查数据，并基于检查结果将文档发送到分片#3。

(3) 分片#3 插入文档。

从客户端的视角来看，这再一次与单服务器设置是完全相同的。

上述场景中配置服务器的角色

思考一下这个场景，你开始得到日期为 2009 年 9 月的数百万个文档的插入请求。在这种情况下，分片#2 会逐渐超负荷。

在配置服务器意识到分片#2 变得过大时，它就会介入。它会划分该分片上的数据并且开始将它迁移到其他分片。在迁移完成之后，它会将更新后的状态发送给 mongos。因此，现在分片#2 具有从 2009 年 8 月到 2009 年 9 月 18 日的数据，而分片#3 包含从 2009 年 9 月 19 日到最晚时间的数据。

当一个新的分片被添加到群集时，配置服务器就要负责计算出用它来做什么。可能需要立即将数据迁移到这个新的分片，或者这个新的分片可能需要留待备用一段时间。总的来说，配置服务器就是大脑。无论何时，有任何数据在分片中移动时，配置服务器就会让 mongos 知道最终的配置，这样一来 mongos 就可以继续进行正确的路由。

7.5.3 数据平衡过程

接下来将了解群集是如何保持平衡的(比如，MongoDB 如何确保所有分片的负荷都均等)。

添加新的数据或迁移已有的数据，或者添加或移除服务器，都会导致数据分布的失衡，这意味着要么一个分片超负荷的具有更多的数据块而其他分片具有较少的数据块，要么它会导致数据块大小的增加，而该数据块的大小明显大于其他数据块。

MongoDB 会使用以下后台程序确保平衡：

- 数据块划分

- 平衡器

1. 数据块划分

数据块划分是其中一个程序，它会确保数据块都是指定大小。如你所见，会选择一个分片键并将其用于指定文档如何跨分片分布。这些文档将进一步被分组成 64MB 的数据块(默认大小且可配置)并且基于它被托管的范围被存储在分片中。

如果由于一个插入或者更新操作导致数据块的大小发生变更，并且超出了默认的数据块大小，那么该数据块就会被 mongos 划分成两个较小的数据块。参见图 7-20。

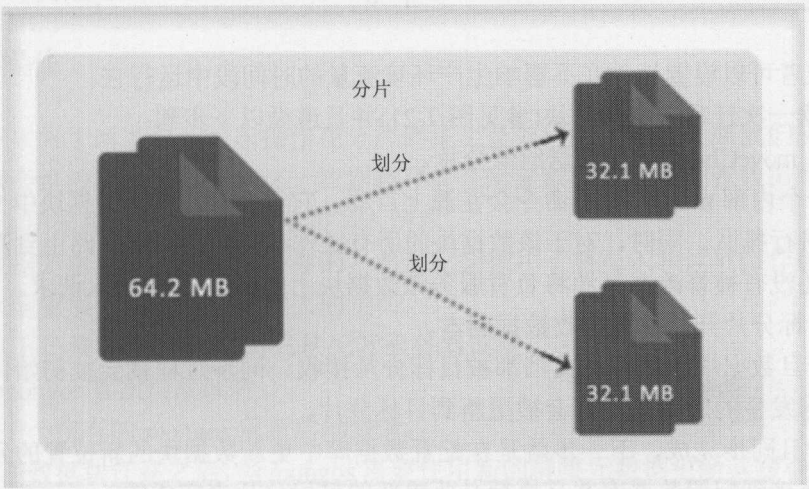

图 7-20 数据块划分

此过程会将数据块保持在指定大小或者小于该指定大小的一个分片中(比如，它会确保数据块都是所配置的大小)。

插入和更新操作会触发划分操作。划分操作会导致配置服务器中的数据修改，因为元数据被修改了。尽管划分操作不会导致数据的迁移，但此操作会导致群集的失衡，其中一个分片的数据块会比另一个分片多。

2. 平衡器

平衡器是后台程序，它被用来确保所有分片的负荷都均等或者处于一种平衡状态。这个程序会管理数据块迁移。

数据块的划分会造成失衡。添加或移除文档也会造成群集失衡。出现群集失衡的情况时就会使用平衡器，它是平均分发数据的程序。

当有一个分片的数据块比其他分片多时，MongoDB 会跨分片自动完成数据块平衡。这一过程对于应用程序和用户来说是透明的。

群集中的任何一个 mongos 都可以初始化平衡器程序。它们可以通过获取配置服务器的配置数据库上的一个锁来达成此目的，因为平衡器涉及从一个分片将数据块迁移到另一个分片，这会导致元数据的变更，从而引发配置服务器数据库中的变更。平衡器程序会对数据库性能造成巨大影响，所以它可以：

(1) 仅在达到迁移阈值时才配置启动迁移。迁移阈值就是分片上最大数据块数量和最小数据块数量的差值。表 7-4 中显示了阈值。

表 7-4 迁移阈值

数据块数量	迁 移 阈 值
< 20	2
21-80	4
>80	8

(2) 或者可以设置计划在不影响生产环境流量的时间段中运行它。

平衡器一次迁移一个数据块(参见图 7-21)并且遵循以下步骤：

(1) 将 moveChunk 命令发送给源分片。

(2) 一个内部 moveChunk 命令会在源上启动，它会在源上创建数据块中文档的副本并且对其进行排队。同时，对于该数据块的所有操作都会被 mongos 路由到源，因为配置数据库还没有被修改并且源将负责服务该数据块上的所有读取/写入请求。

(3) 目标分片开始从源接收数据副本。

(4) 一旦数据块中所有的文档都被目标分片接收，同步过程就会被初始化以确保迁移期间数据发生的所有变更都会被更新到目标分片。

(5) 一旦同步完成，下一步就是在配置数据库中更新数据块的新位置的元数据。此操作是由连接到配置数据库并且执行必要更新的目标分片来完成的。

(6) 在上述所有步骤成功完成之后，在源分片上维护的文档副本就会被删除。

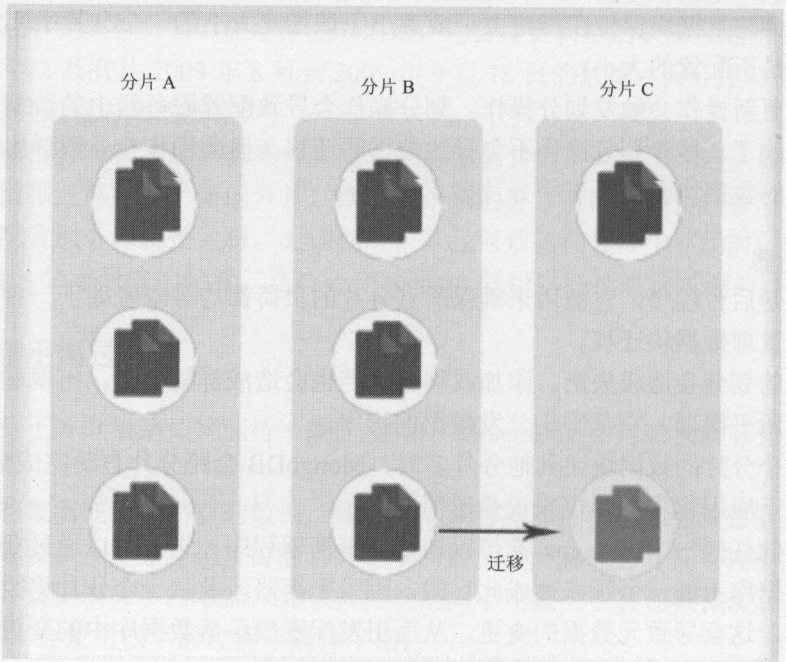

图 7-21 数据块迁移

如果平衡器同时需要从源分片进行额外的数据块迁移，则它可以开启新的迁移，而甚至无须等待当前迁移的删除步骤完成。

如果启用过程期间发生任何错误，平衡器就会终止该过程，将数据块留在原分片上。在该过程成功完成时，MongoDB 会从原分片上移除该数据块的数据。

添加或移除分片也会导致群集失衡。在添加一个新的分片时，会立即开始对该分片的数据迁移。不过，群集平衡需要一些时间。

在移除一个分片时，平衡器会确保数据被迁移到其他分片并且更新元数据信息。在这两个操作完成之后，该分片会被安全移除。

7.5.4 操作

接下来将了解读取和写入操作在分片群集上是如何执行的。正如我们所提及的，配置服务器会维护群集元数据。这一数据被存储在配置数据库中。配置数据库的这一数据被 mongos 用来服务应用程序读取和写入请求。

该数据是由 mongos 实例缓存的，然后它会被用于将写入和读取操作路由到分片。这样一来配置服务器就不会有过大的负荷。

在以下场景中，mongos 将只从配置服务器读取：

- mongos 首次启动或者；
- 一个现有的 mongos 重新启动或者；
- 在数据块迁移之后，当 mongos 需要使用新的群集元数据更新其缓存的元数据时。

无论何时执行了任何操作，mongos 需要做的第一步都是识别出要服务该请求的分片。由于分片键被用于跨分片群集分发数据，因此，如果该操作使用分片键字段，那么基于此就可以定位出特定的分片。

如果分片键是 employeeid，那么会发生以下事情：

(1) 如果 find 查询包含 employeeid 字段，那么为了满足该查询，就只有特定的分片才会被 mongos 定位出来。

(2) 如果单个更新操作将 employeeid 用于更新文档，则该请求会被路由到持有该员工数据的分片。

不过，如果该操作没有使用分片键，那么该请求会被广播到所有分片。通常多项更新或移除操作都是跨群集来定位的。

在查询数据时，可能会出现这种情况，除了识别分片和从中获得数据之外，mongos 可能还需要在将最终输出结果发送给客户端之前处理由各个分片返回的数据。

假设一个应用程序已经发出了带有 sort()的 find()请求。在这种情况下，mongos 就会将$orderby 操作传递给分片。分片会从其数据集中抓取数据并且以排序的方式发送结果。一旦 mongos 具有所有分片的排序数据，它就会对所有数据执行递增的合并排序，然后将最终输出结果返回给客户端。

像 limit()、skip()等这样的聚合函数类似于 sort，它们需要 mongos 在从分片接收到数据之后并且在将最终结果集返回给客户端之前执行操作。

mongos 会消耗最小的系统资源并且没有持久化状态。因此，如果应用程序的需求是一个简单的 find()查询，分片就可以满足它并且无需 mongos 级别的操作，那么可以在应用服务器运行的相同系统上运行 mongos。

7.5.5 实现分片

在本节中，将学习在 Windows 平台的一台机器上配置分片。

通过仅使用两个分片，让该示例保持为简单水平。在此配置中，要使用表 7-5 中列出的服务。

表 7-5 分片群集配置

组件	类型	端口	数据文件路径
分片控制器	Mongos	27021	-
配置服务器	Mongod	27022	C:\db1\config\data
Shard0	Mongod	27023	C:\db1\shard1\data
Shard1	Mongod	27024	C:\db1\shard2\data

你将专注于以下内容：

(1) 设置一个分片群集。

(2) 创建一个数据库和集合，并且在集合上启用分片。

(3) 使用导入命令在分片集合中加载数据。

(4) 在分片之间分发数据。

(5) 从群集添加和移除分片，并且检查数据是如何自动分发的。

1. 设置分片群集

为了设置群集，第一步是设置配置服务器。在一个新的终端窗口中输入以下代码，以便为配置服务器创建数据目录并且开启 mongod：

```
C:\> mkdir C:\db1\config\data
C:\>CD C:\practicalmongodb\bin
C:\ practicalmongodb\bin>mongod --port 27022 --dbpath C:\db1\config\data
--configsvr

2015-07-13T23:02:41.982-0700 I JOURNAL [journal writer] Journal writer
thread started
2015-07-13T23:02:41.984-0700 I CONTROL [initandlisten] MongoDB starting :
pid=3084
```

```
    port=27022 dbpath=C:\db1\config\data master=1 64-bit host=ANOC9
    ..................................
    2015-07-13T23:02:42.066-0700 I REPL [initandlisten] ******
    2015-07-13T03:02:42.067-0700 I NETWORK [initandlisten] waiting for
connections on port 27022
```

接下来，开启 mongos。在一个新的终端窗口中输入以下代码：

```
    C:\>cd c:\practicalmongodb\bin
    c:\practicalmongodb\bin>mongos --configdb localhost:27022 --port 27021
--chunkSize 1
    2015-07-13T23:06:07.246-0700 W SHARDING running with 1 config server
should be done only for
    testing purposes and is not recommended for production
    ..............................................................
    2015-07-13T23:09:07.464-0700 I SHARDING [Balancer] distributed lock
'balancer/
    ANOC9:27021:1429783567:41' unlocked
```

现在启动了分片控制器(比如 mongos)并且让它运行起来了。

如果切换到启动了配置服务器的窗口，就会看到该分片服务器注册到配置服务器的信息。

在这个示例中，你已经使用了 1MB 的数据块大小。注意，这在现实场景中并非理想值，因为其大小小于 4MB(一个文档的最大大小)。不过，这仅是用于说明而已，因为这会创建必要的数据块数量，而无须加载大量数据。默认情况下，chunkSize 是 128MB，除非另行指定。

接下来，启动分片服务器，Shard0 和 Shard1。

打开一个新的终端窗口。为第一个分片创建数据目录并且启动 mongod：

```
    C:\>mkdir C:\db1\shard0\data
    C:\>cd c:\practicalmongodb\bin
    c:\practicalmongodb\bin>mongod --port 27023 --dbpath c:\db1\shard0\data
-shardsvr
    2015-07-13T23:14:58.076-0700 I CONTROL [initandlisten] MongoDB starting :
pid=1996 port=27023 dbpath=c:\db1\shard0\data 64-bit host=ANOC9
    ..............................................................
    2015-07-13T23:14:58.158-0700 I NETWORK [initandlisten] waiting for
connections on port 27023
```

打开一个新的终端窗口。为第二个分片创建数据目录并且启动 mongod：

```
    C:\>mkdir c:\db1\shard1\data
    C:\>cd c:\practicalmongodb\bin
    c:\practicalmongodb\bin>mongod --port 27024 --dbpath C:\db1\shard1\data
--shardsvr
    2015-07-13T23:17:01.704-0700 I CONTROL [initandlisten] MongoDB starting :
```

```
pid=3672 port=27024 dbpath=C:\db1\shard1\data 64-bit host=ANOC9
    2015-07-13T23:17:01.704-0700 I NETWORK [initandlisten] waiting for
connections on port 27024
```

在上述步骤结束后，与该设置有关的所有服务器都启动并且运行了。下一步是将分片信息添加到分片控制器。

对于应用程序来说，mongos 表现得就像一个完整的 MongoDB 实例一样，尽管实际上它并非一个完整的实例。mongo shell 可被用于连接到 mongos 以在其上执行任何操作。

打开 mongos mongo 控制台：

```
C:\>cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos>
```

切换到 admin 数据库：

```
mongos> use admin
switched to db admin
mongos>
```

通过运行以下命令来添加分片信息：

```
mongos> db.runCommand({addshard:"localhost:27023",allowLocal:true})
{ "shardAdded" : "shard0000", "ok" : 1 }
mongos> db.runCommand({addshard:"localhost:27024",allowLocal:true})
{ "shardAdded" : "shard0001", "ok" : 1 }
mongos>
```

这样就启动了这两台分片服务器。

下一个命令会检查这两个分片：

```
mongos> db.runCommand({listshards:1})
{
        "shards" : [
                {
                        "_id" : "shard0000",
                        "host" : "localhost:27023"
                }, {
                        "_id" : "shard0001",
                        "host" : "localhost:27024"
                }
        ],      "ok" : 1}
```

2. 创建一个数据库和分片集合

为了使用这个示例继续进一步讲解，你要创建一个名称为 testdb 的数据库和一个名称为 testcollection 的集合，你要基于键 testkey 对其分片。

连接到 mongos 控制台并且运行以下命令来得到该数据库：

```
mongos> testdb=db.getSisterDB("testdb")
testdb
```

接着，在数据库级别为 testdb 启用分片：

```
mongos> db.runCommand({enableSharding system: "testdb"})
{ "ok" : 1 }
mongos>
```

接下来，指定需要被分片的集合以及基于哪个键来对该集合分片：

```
mongos> db.runCommand({shardcollection: "testdb.testcollection", key:
{testkey:1}})
{ "collectionsharded" : "testdb.testcollection", "ok" : 1 }
mongos>
```

在上述步骤完成之后，你就有了一个设置好的分片群集，其所有组件都已经启动并且运行了。你还创建了一个数据库并且对集合启用了分片。

接下来，将数据导入到该集合，以便可以检查分片上分布的数据。

要使用导入命令将数据加载到 testcollection。连接到一个新的终端窗口并且执行以下命令：

```
C:\>cd C:\practicalmongodb\bin
C:\practicalmongodb\bin>mongoimport --host ANOC9 --port 27021 --db testdb
--collection
testcollection --type csv --file c:\mongoimport.csv --headerline
2015-07-13T23:17:39.101-0700 connected to: ANOC9:27021
2015-07-13T23:17:42.298-0700 [###############.........]
testdb.testcollection 1.1 MB/1.9 MB (59.6%)
2015-07-13T23:17:44.781-0700 imported 100000 documents
```

mongoimport.csv 由两个字段构成。第一个是 testkey，它是一个随机生成的数字。第二个字段是一个文本字段；它被用于确保文档占据足够数量的数据块，让其可以使用分片机制。

这会将 100 000 个对象插入到该集合中。

为了验证这些记录是否被插入了，需要连接到 mongos 的 mongo 控制台并且运行以下命令：

```
C:\Windows\system32>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
```

```
connecting to: localhost:27021/test
mongos> use testdb
switched to db testdb
mongos> db.testcollection.count()
100000
mongos>
```

接下来，连接到这两个分片(Shard0 和 Shard1)的控制台，并且查看数据是如何分发的。打开一个新的终端窗口并且连接到 Shard0 的控制台：

```
C:\>cd C:\practicalmongodb\bin
C:\ practicalmongodb\bin>mongo localhost:27023
MongoDB shell version: 3.0.4
connecting to: localhost:27023/test
```

切换到 testdb 并且运行 count()命令以检查该分片上文档的数量：

```
> use testdb
switched to db testdb
>db.testcollection.count()
57998
```

接下来，打开一个新的终端窗口，连接到 Shard1 的控制台，并遵循上述步骤(比如，切换到 testdb 并且检查 testcollection 集合的文档数)：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27024
MongoDB shell version: 3.0.4
connecting to: localhost:27024/test
> use testdb
switched to db testdb
>db.testcollection.count()
42002
>
```

有时当运行上述命令时，可能会发现每个分片中文档数量有差异。在加载文档时，所有的数据块都会被 mongos 放置到一个分片上。经过一段时间后，会通过跨所有分片平均分发数据块使该分片集重新平衡。

---

3. 添加一个新分片

你设置好了一个分片群集，并且也对一个集合进行了分片以及查看了数据是如何在分片之间分布的。接下来，要将一个新的分片添加到该群集以便负荷更进一步地分散开来。

要重复上面提及的步骤。首先在一个新的终端窗口中为这个新的分片创建一个数据目录：

```
c:\>mkdir c:\db1\shard2\data
```

接下来，开启端口 27025 上的 mongod：

```
c:\>cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongod --port 27025 --dbpath C:\db1\shard2\data
--shardsvr
2015-07-13T23:25:49.103-0700 I CONTROL [initandlisten] MongoDB starting :
pid=3744 port=27025 dbpath=C:\db1\shard2\data 64-bit host=ANOC9
.....
2015-07-13T23:25:49.183-0700 I NETWORK [initandlisten] waiting for
connections on port 27025
```

接下来，这个新的分片服务器会被添加到分片群集。为了配置它，你要在一个新的终端窗口中打开其 mongos mongo 控制台：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos>
```

切换到 admin 数据库并且运行 addshard 命令。这个命令会将该分片服务器添加到分片群集。

```
mongos> use admin
switched to db admin
mongos> db.runCommand({addshard: "localhost:27025", allowlocal: true})
{ "shardAdded" : "shard0002", "ok" : 1 }
mongos>
```

为了验证是否成功添加，要运行 listshards 命令：

```
mongos> db.runCommand({listshards:1})
{
        "shards" : [
                {
                        "_id" : "shard0000",
                        "host" : "localhost:27023"
                },
                {
                        "_id" : "shard0001",
                        "host" : "localhost:27024"
                },
                {
                        "_id" : "shard0002",
                        "host" : "localhost:27025"
                }
```

```
        ],
        "ok" : 1
}
```

接下来，检查 testcollection 数据是如何分布的。在一个新的终端窗口中连接到这个新分片的控制台：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27025
MongoDB shell version: 3.0.4
connecting to: localhost:27025/test
```

切换到 testdb 并且检查该分片上列出的集合：

```
> use testdb
switched to db testdb
> show collections
system.indexes
testcollection
```

运行三次 testcollection.count 命令：

```
>db.testcollection.count()
6928
>db.testcollection.count()
12928
>db.testcollection.count()
16928
```

有意思的是，该集合中条目的数量在缓慢增长。mongos 正在重新平衡该群集。

随着时间的推移，会将数据块从分片服务器 Shard0 和 Shard1 迁移到新添加的分片服务器 Shard2，这样一来数据就会跨所有服务器平均分布。在这一过程完成之后，就会更新配置服务器元数据。这是一个自动化的过程，即使 testcollection 中没有添加新数据，该过程也会发生。这是你在决定数据块大小时需要考虑的其中一个重要因素。

如果 chunkSize 的值非常大，那么数据最终将分布得不那么平均。当 chunkSize 较小时，数据会分布得更加平均。

4. 移除一个分片

在下面的示例中，将看到如何移除一台分片服务器。对于这个示例，你要移除在上面示例中添加的服务器。

为了初始化该过程，需要登录到 mongos 控制台，切换到 admin db，并且执行以下命令从分片群集中移除该分片：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
```

```
connecting to: localhost:27021/test
mongos> use admin
switched to db admin
mongos> db.runCommand({removeShard: "localhost:27025"})
{
        "msg" : "draining started successfully",
        "state" : "started",
        "shard" : "shard0002",
        "ok" : 1
}
mongos>
```

如你所见，removeShard 命令会返回一条消息。该消息的其中一个字段是 state，它表明了处理过程状态。该消息还表明，数据清理过程已经开始。这是由字段 msg 表示的。

可以运行 removeShard 命令来检查该过程：

```
mongos> db.runCommand({removeShard: "localhost:27025"})
{
        "msg" : "draining ongoing",
        "state" : "ongoing",
        "remaining" : {
                "chunks" : NumberLong(2),
                "dbs" : NumberLong(0)
        },
        "ok" : 1
}
mongos>
```

其响应告诉你仍然需要从服务器上清理的数据块和数据库的数量。如果重新运行该命令并且其过程已经结束，那么该命令的输出结果将描述相同内容。

```
mongos> db.runCommand({removeShard: "localhost:27025"})
{
        "msg" : "removeshard completed successfully",
        "state" : "completed",
        "shard" : "shard0002",
        "ok" : 1
}
mongos>
```

可以使用 listshards 来验证 removeShard 是否成功。

如你所见，数据被成功迁移到了其他分片，因此可以删除存储文件并且终止 Shard2 mongod 程序。

此无需离线就能修改分片群集的功能是 MongoDB 的其中一个关键组件，它使得 MongoDB 可以支持高可用性、高可扩展性以及大容量的数据存储。

---

5. 列示分片群集状态

printShardingStatus()命令会给出分片系统内部的大量有用信息。

```
mongos> db.printShardingStatus()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "version" : 3,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("52fb7a8647e47c5884749a1a")
}
  shards:
        { "_id" : "shard0000", "host" : "localhost:27023" }
        { "_id" : "shard0001", "host" : "localhost:27024" }
balancer:
        Currently enabled: yes
        Currently running: no
        Failed balancer rounds in last 5 attempts: 0
        Migration Results for the last 24 hours:
                17 : Success
databases:
        { "_id" : "admin", "partitioned" : false, "primary" : "config" }
        { "_id" : "testdb", "partitioned" : true, "primary" : "shard0000" }
  ..............
```

该输出结果列出了以下信息：

- 分片群集的所有分片服务器
- 每个分片数据库/集合的配置
- 分片数据集的所有数据块

可以从上面的命令中获得的重要信息是分片键范围，它与每一个数据块有关。这也表明了特定数据块被存储在何处(在哪台分片服务器上)。该输出结果可以被用来分析分片服务器的键以及数据块分布。

7.5.6 控制集合分布(基于标签分片)

在上一节中，你看到了数据分发如何进行。在本节中，将学习与基于标签分片有关的内容。这一功能是在版本 2.2.0 中引入的。

标签让操作者可以控制将哪些集合放到哪个分片。

为了理解基于标签分片，我们来设置一个分片群集。将使用前面创建的分片群集。对于本示例，需要 3 个分片，因此你要再次将 Shard2 添加到该群集。

1. 先决条件

首先要启动该群集。再重申一次，执行这些步骤。

(1) 开启配置服务器。在一个新的终端窗口中输入以下命令(如果它还没有运行的话)：

```
C:\> mkdir C:\db1\config\data
C:\>cd c:\practicalmongodb\bin
C:\practicalmongodb\bin>mongod --port 27022 --dbpath C:\db\config\data
--configsvr
```

(2) 开启 mongos。在一个新的终端窗口中输入以下命令(如果它还没有运行的话)：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongos --configdb localhost:27022 --port 27021
```

(3) 接下来启动分片服务器。

启动 Shard0。在一个新的终端窗口中输入以下命令(如果它还没有运行的话)：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongod --port 27023 --dbpath c:\db1\shard0\data
--shardsvr
```

启动 Shard1。在一个新的终端窗口中输入以下命令(如果它还没有运行的话)：

```
C:\>cd c:\practicalmongodb\bin
C:\practicalmongodb\bin>mongod --port 27024 --dbpath c:\db1\shard1\data
-shardsvr
```

启动 Shard2。在一个新的终端窗口中输入以下命令(如果它还没有运行的话)：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongod --port 27025 --dbpath c:\db1\shard2\data
-shardsvr
```

因为在之前的示例中已经将 Shard2 从分片群集中移除了，所以必须将 Shard2 添加到分片群集，因为对于这个示例来说，需要三个分片。

为此，需要连接到 mongos。输入以下命令：

```
C:\Windows\system32>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0. 4
connecting to: localhost:27021/test
mongos>
```

在将该分片添加到群集之前，需要删除 testdb 数据库：

```
mongos> use testdb
switched to db testdb
```

```
mongos> db.dropDatabase()
{ "dropped" : "testdb", "ok" : 1 }
mongos>
```

接下来，使用以下步骤添加 Shard2 分片：

```
mongos> use admin
switched to db admin
mongos> db.runCommand({addshard: "localhost:27025", allowlocal: true})
{ "shardAdded" : "shard0002", "ok" : 1 }
mongos>
```

如果尝试添加已经移除的分片而不移除 testdb 数据库，则会得到以下错误：

```
mongos>db.runCommand({addshard: "localhost:27025", allowlocal: true})
{
        "ok" : 0,
        "errmsg" : "can't add shard localhost:27025 because a local
    database 'testdb' exists in another shard0000:localhost:27023"}
```

为了确保所有 3 个分片都存在于群集中，需要运行以下命令：

```
mongos> db.runCommand({listshards:1})
{
        "shards" : [
                {
                        "_id" : "shard0000",
                        "host" : "localhost:27023"
                }, {
                        "_id" : "shard0001",
                        "host" : "localhost:27024"
                }, {
                        "_id" : "shard0002",
                        "host" : "localhost:27025"
                }
        ], "ok" : 1}
```

2. 标签

在上述步骤结束时，你就有了启动且运行一台配置服务器、3 个分片以及一个 mongos 的分片群集。接下来，在一个新的终端窗口中连接到 30999 端口上的 mongos 和 27022 上的 configdb：

```
C:\ >cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongos --port 30999 --configdb localhost:27022
2015-07-13T23:24:39.674-0700 W SHARDING running with 1 config server
should be done only for testing purposes and is not recommended for production
```

```
.....................................
    2015-07-13T23:24:39.931-0700 I SHARDING [Balancer] distributed lock
'balancer /ANOC9:30999:1429851279:41' unlocked..
```

接着，开启一个新的终端窗口，连接到 mongos，并且在集合上启用分片：

```
C:\ >cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo localhost:27021
MongoDB shell version: 3.0.4
connecting to: localhost:27021/test
mongos> show dbs
admin (empty)
config 0.016GB
testdb 0.078GB
mongos> conn=new Mongo("localhost:30999")
connection to localhost:30999
mongos> db=conn.getDB("movies")
movies
mongos> sh.enableSharding("movies")
{ "ok" : 1 }
mongos> sh.shardCollection("movies.drama", {originality:1})
{ "collectionsharded" : "movies.hindi", "ok" : 1 }
mongos> sh.shardCollection("movies.action", {distribution:1})
{ "collectionsharded" : "movies.action", "ok" : 1 }
mongos> sh.shardCollection("movies.comedy", {collections:1})
{ "collectionsharded" : "movies.comedy", "ok" : 1 }
mongos>
```

其步骤如下：

(1) 连接到 mongos 控制台。

(2) 浏览连接到运行在端口 30999 上 mongos 实例的运行中的数据库。

(3) 得到数据库 movies 的引用。

(4) 启用数据库 movies 的分片。

(5) 通过分片键 originality 对集合 movies.drama 分片。

(6) 通过分片键 distribution 对集合 movies. action 分片。

(7) 通过分片键 collections 对集合 movies. comedy 分片。

接下来，使用以下一组命令将一些数据插入到集合中：

```
    mongos>for(var i=0;i<100000;i++){db.drama.insert({originality:Math.
random(), count:i, time:new Date()});}
    mongos>for(var i=0;i<100000;i++){db.action.insert({distribution:Math.
random(), count:i, time:new Date()});}
    mongos>for(var i=0;i<100000;i++) {db.comedy.insert({collections:Math.
random(), count:i,
    time:new Date()});}
```

```
mongos>
```

在上述步骤结束时，你就有了 3 个分片和 3 个集合，并且对这些集合启用了分片。接着你要查看数据是如何跨分片分布的。

切换到 configdb：

```
mongos> use config
switched to db config
mongos>
```

可以使用 chunks.find 来查看数据块是如何分布的：

```
mongos> db.chunks.find({ns:"movies.drama"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos> db.chunks.find({ns:"movies.action"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos> db.chunks.find({ns:"movies.comedy"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
```

```
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos>
```

如你所见，这些数据块是正好在分片之间平均分布的。参见图 7-22。

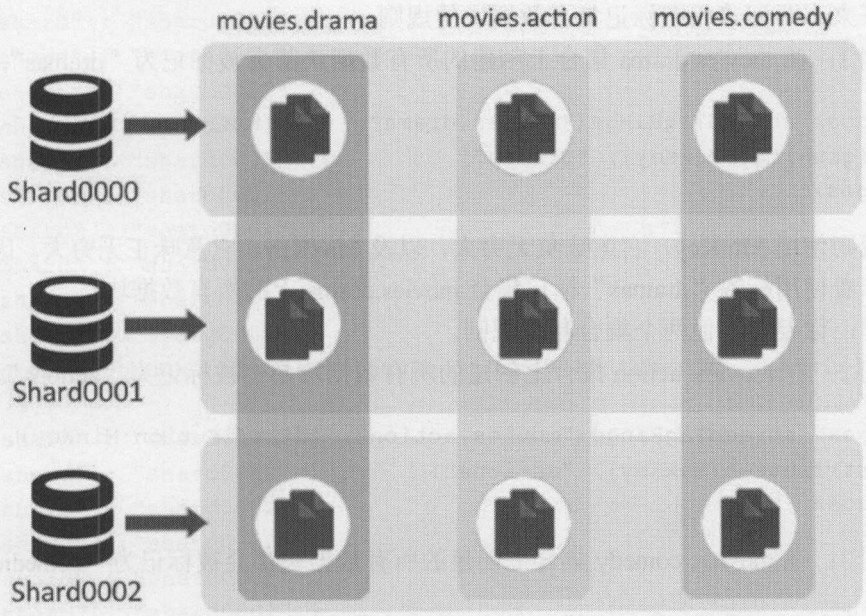

图 7-22 不使用标签的分布

接下来，你要使用标签来分离集合。这样做的目的是让一个分片具有一个集合(比如，你的目标是得到表 7-6 中所示的数据块分布)。

表 7-6 数据块分布

集合数据块	所处的分片
movies.drama	Shard0000
movies.action	Shard0001
movies.comedy	Shard0002

标签描述了分片的属性，它可以是任何内容。因此可以将一个分片标记为“慢”或“快”或“机架空间”或“西海岸”。

在以下示例中，你要将分片标记为归属于单个集合：

```
mongos> sh.addShardTag("shard0000", "dramas")
mongos> sh.addShardTag("shard0001", "actions")
mongos> sh.addShardTag("shard0002", "comedies")
mongos>
```

这意味着以下几点：

- 将标签为“dramas”的数据块放置在 shard0000。
- 将标签为“actions”的数据块放置在 shard0001。
- 将标签为“comedies”的数据块放置在 shard0002。

接下来，要创建相应标记集合数据块的规则。

规则 1：在 movies.drama 集合上创建的所有数据块都会被标记为“dramas”：

```
mongos> sh.addTagRange("movies.drama", {originality:MinKey},
{originality:MaxKey}, "dramas")
mongos>
```

该规则使用 MinKey，它意味负无穷大，以及 MaxKey，它意味正无穷大。因此上述规则意味着使用标签“dramas”标记集合 movies.drama 上的所有数据块。

同样，你要为其他两个集合制定规则。

规则 2：在 movies. action 集合上创建的所有数据块都会被标记为“actions”。

```
mongos> sh.addTagRange("movies.action", {distribution:MinKey},
{distribution:MaxKey}, "actions")
mongos>
```

规则 3：在 movies. comedy 集合上创建的所有数据块都会被标记为“comedies”。

```
mongos> sh.addTagRange("movies.comedy", {collection:MinKey},
{collection:MaxKey}, "comedies")
mongos>
```

需要等待群集重新平衡，以便数据块基于上面定义的标签和规则分布。正如所提及的，数据块分布是一个自动化过程，因此在一段时间之后，数据块将被自动重新分发以实现你做出的变更。

接下来，运行 chunks.find 来验证数据块组织结构：

```
mongos> use config
switched to db config
mongos> db.chunks.find({ns:"movies.drama"}, {shard:1, _id:0}).sort
({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
```

```
mongos> db.chunks.find({ns:"movies.action"}, {shard:1, _id:0}).sort
({shard:1})
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
mongos> db.chunks.find({ns:"movies.comedy"}, {shard:1, _id:0}).sort
({shard:1})
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
mongos>
```

因此，集合数据块已经基于所定义的标签和规则重新分布了，参见图 7-23。

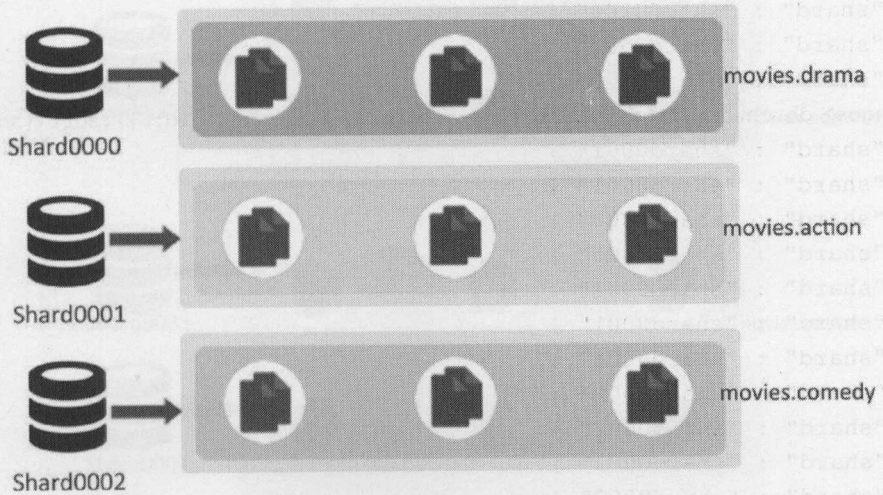

图 7-23　使用标签的分布

3. 使用标签进行扩展

接下来，你要查看如何使用标签进行扩展。我们转换一下场景。假设集合 movies.action 需要两台服务器用于其数据。由于你只有 3 个分片，这意味着其他两个集合的数据需要被移动到一个分片。

在这种情况下，你要修改分片的标签。你要将标签“comedies”添加到 Shard0 并且从 Shard2 移除该标签，并且进一步将标签“actions”添加到 Shard2。

这意味着标记为“comedies”的数据块将被移动到 Shard0，而标记为“actions”的数据块将被分发到 Shard2。

你首先要将集合 movies.comedy 的数据块移动到 Shard0，并且从 Shard2 移除相同的数据块：

```
mongos> sh.addShardTag("shard0000","comedies")
mongos> sh.removeShardTag("shard0002","comedies")
```

接下来，要将标签“actions”添加到 Shard2，这样一来 movies.action 数据块也就会分布到 Shard2：

```
mongos> sh.addShardTag("shard0002","actions")
```

在一段时间之后重新运行 find 命令将会显示以下结果：

```
mongos> db.chunks.find({ns:"movies.drama"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
mongos> db.chunks.find({ns:"movies.action"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0001" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
{ "shard" : "shard0002" }
```

```
{ "shard" : "shard0002" }
mongos> db.chunks.find({ns:"movies.comedy"}, {shard:1, _id:0}).sort({shard:1})
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
{ "shard" : "shard0000" }
mongos>
```

已经重新分发的数据块反映了所做的变更，参见图 7-24。

4. 多标签

可以将多个标签与分片关联。我们将两个不同的标签添加到分片。

假设你希望基于硬盘分发写操作。你有一个分片具有一个旋转磁盘式硬盘，而其他的具有一个 SSD(固态硬盘)。你希望将 50%的写操作导向到使用 SSD 的分片，并将剩余的导向到使用旋转磁盘式硬盘的分片。

首先，基于这些属性标记分片：

```
mongos> sh.addShardTag("shard0001", "spinning")
mongos> sh.addShardTag("shard0002", "ssd")
mongos>
```

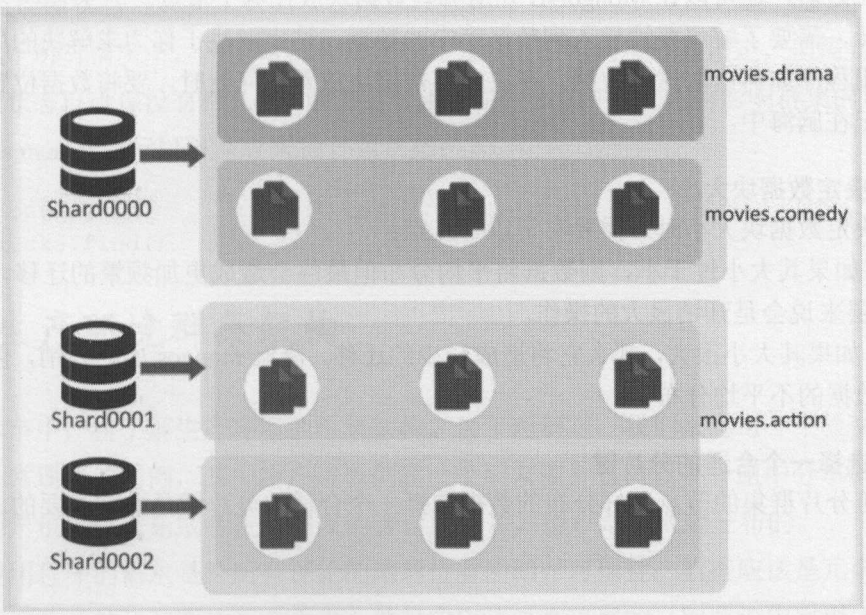

图 7-24 使用标签进行扩展

我们进一步假设你的 movies.action 集合有一个 distribution 字段，你要将它用作分片键。该 distribution 字段值介于 0 和 1 之间。接着，你希望，“如果 distribution<0.5，则将文档发送到旋转磁盘式硬盘。如果 distribution≥0.5，则发送到 SSD。”所以你要定义如下规则：

```
    mongos>sh.addTagRange("movies.action", {distribution:MinKey} ,
{distribution:.5} ,"spinning")
    mongos>sh.addTagRange("movies.action" ,{distribution:.5} ,{distribution
:MaxKey},"ssd")
    mongos>
```

现在 distribution<.5 的文档将被写到旋转磁盘式硬盘分片，而其他的被写到 SSD 硬盘分片。

使用标签你就可以控制每台新添加服务器将会得到的负荷类型。

7.5.7 在将数据导入到分片环境时要记住的要点

这里是在导入数据时要牢记的一些要点。

1. 数据的预划分

相较于留待 MongoDB 进行数据块创建的选择，可以使用以下命令告知 MongoDB 如何进行数据块创建：

```
    db.runCommand( { split : "practicalmongodb.mycollection" , middle :
{ shardkey : value } } );
```

在此之后，还可以让 MongoDB 知道哪些数据块放在哪个节点。

为此，需要了解将要被导入到数据库中的数据。这也取决于你力求解决的用例以及你的应用程序如何读取这些数据。在决定将数据块放置在何处时，要将数据位置这样的信息牢记在脑海中。

2. 决定数据块大小

在决定数据块大小时，需要牢记以下几点：

(1) 如果其大小过于小，则数据将平均分布但最终会造成更加频繁的迁移，这对于 mongos 层来说会是开销极大的操作。

(2) 如果其大小很大，那么它将造成较少的迁移，降低 mongos 层的开销，但你最终会面临数据的不平均分布。

3. 选择一个合适的分片键

为跨分片群集的节点良好分布的数据选择一个合适的分片键是非常必要的。

7.5.8 监控分片

除了用于其他 MongoDB 实例的常规监控和分析之外，分片群集还需要额外的监控来确保其所有操作都运行正常并且数据在节点之间有效分布。在本节中，将看到应该为分片群集的正常运行进行哪些监控。

7.5.9 监控配置服务器

正如你目前所知道的，配置服务器会存储分片群集的元数据。mongos 会缓存这些数据并且将请求路由到各自的分片。如果配置服务器宕机但有一个运行中的 mongos 实例，那么这不会立即对分片群集造成影响，它仍将保持一段时间的可用状态。不过，将无法执行像数据块迁移或重启一个新的 mongos 这样的操作。从长远来看，配置服务器的不可用会对群集的可用性造成严重影响。为了确保群集仍旧保持平衡且可用，应该监控配置服务器。

1. 监控分片状态平衡和数据块分布

为了得到最有效地分片群集部署，需要数据块在分片之间平均分布。正如你目前所知道的，这是由 MongoDB 使用一个后台程序自动完成的。需要监控分片状态来确保该程序正在有效工作。为此，可以在 mongos mongo shell 中使用 db.printShardingStatus()或 sh.status()命令来确保该程序正在有效工作。

2. 监控锁状态

在几乎所有情况下，平衡器都会在完成其处理之后自动释放其锁定，但需要检查数据库的锁状态以确保没有持久锁，因为这会阻塞进一步的平衡，它将影响群集的可用性。从 mongos mongo 运行以下命令来检查锁状态：

```
use config
db.locks.find()
```

7.6 生产环境群集架构

在本节中，将了解生产环境的群集架构。为了理解它，我们来思考一个社交网络应用程序非常通用的用例，其中用户可以创建一个朋友圈并且可以在分组中分享其评论或图片。用户也可以评论或者赞其朋友的评论或图片。用户是呈地理分布的。

该应用程序的需求是跨所有评论的地理位置的即时可用性；数据应该是冗余的，以便用户的评论、帖子和图片不会丢失；并且它应该是高可用的。因此该应用程序的生产

环境群集应该使用以下组件：

(1) 至少两个 mongos 实例，但可以根据需要使用更多个。

(2) 3 台配置服务器，每台服务器位于一个单独的系统上。

(3) 两个或更多个副本集充当分片。这些副本集是跨地理位置分布的，具有设置为 nearest 的读关注。参见图 7-25。

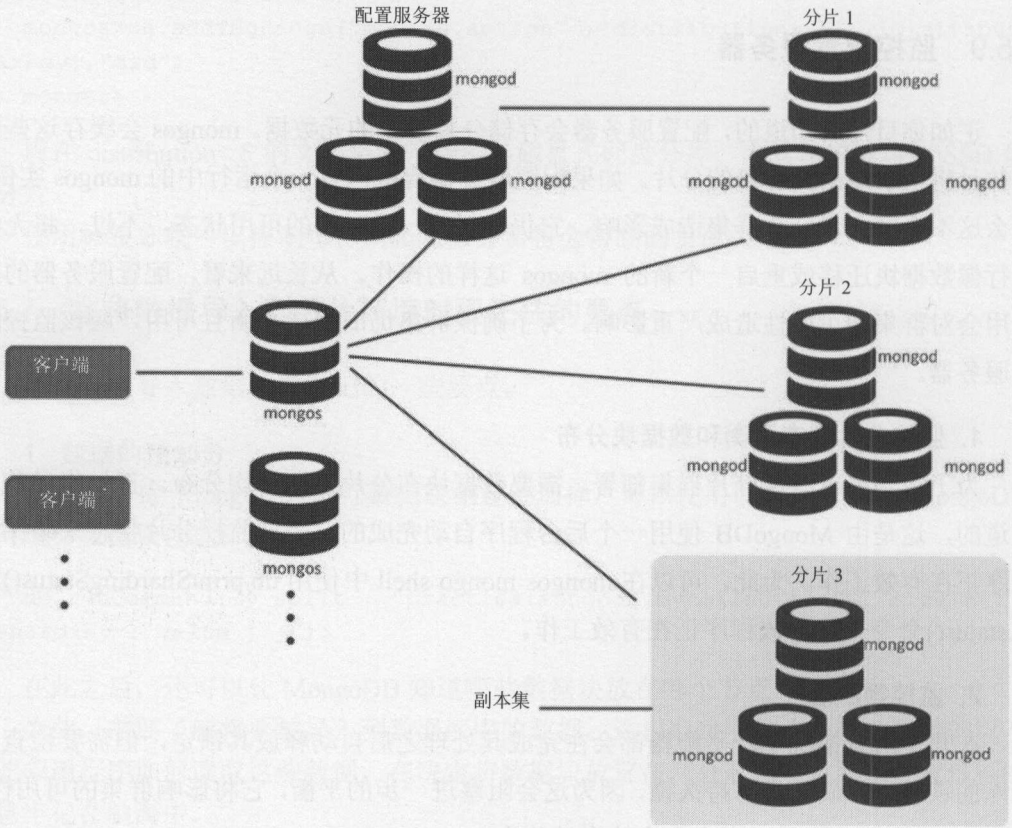

图 7-25 生产环境群集架构

现在我们来看看 MongoDB 生产环境部署中可能的故障场景以及其对于环境的影响。

7.6.1 场景 1

Mongos 变得不可用：mongos 已经关闭的应用程序服务器将无法与群集通信，但这不会导致任何数据丢失，因为 mongos 不会维护其自身的任何数据。mongos 可以重启，并且在重启期间，它可以与配置服务器同步以缓存群集元数据，而应用程序可以正常启动其操作，参见图 7-26。

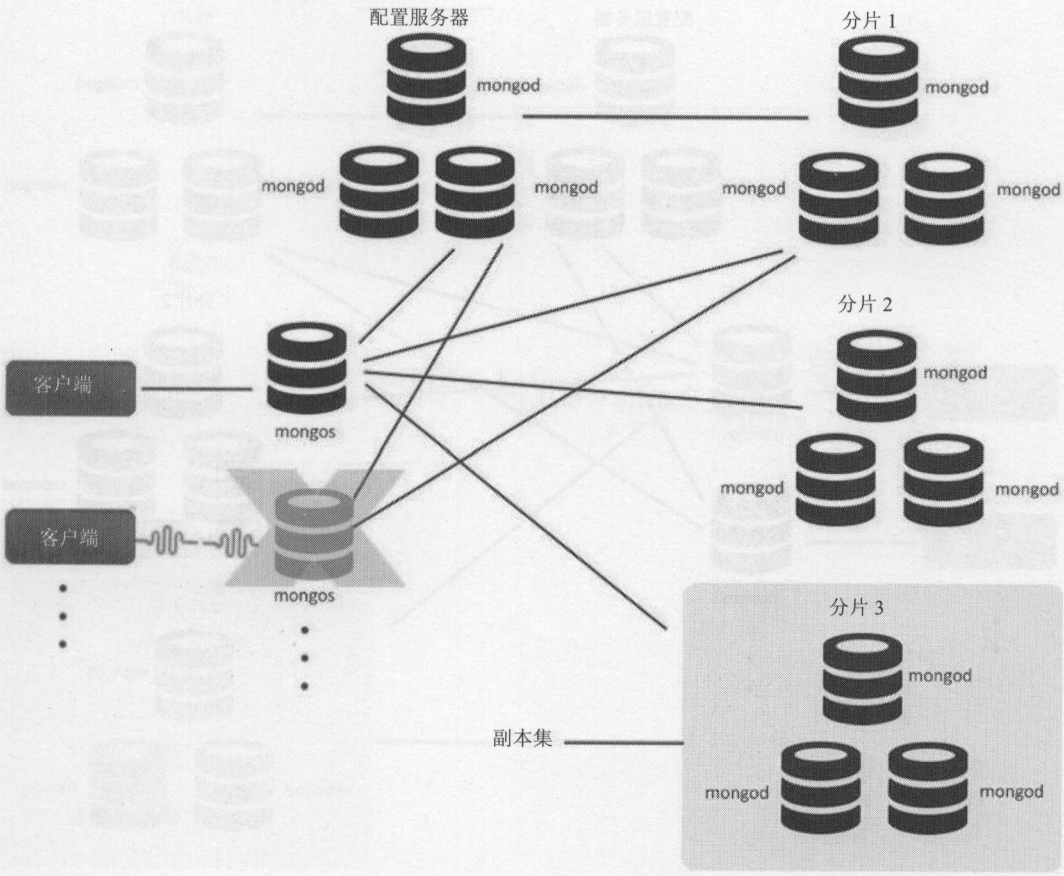

图 7-26 Mongos 变得不可用

7.6.2 场景 2

一个分片中其中一个副本集的 mongod 变得不可用：由于你使用了副本集来提供高可用性，因此不会存在数据丢失的情况。如果主节点关闭，则会选择一个新的主节点，而如果一个辅助节点关闭，那么它会断开连接，而功能将继续保持正常，如图 7-27 所示。

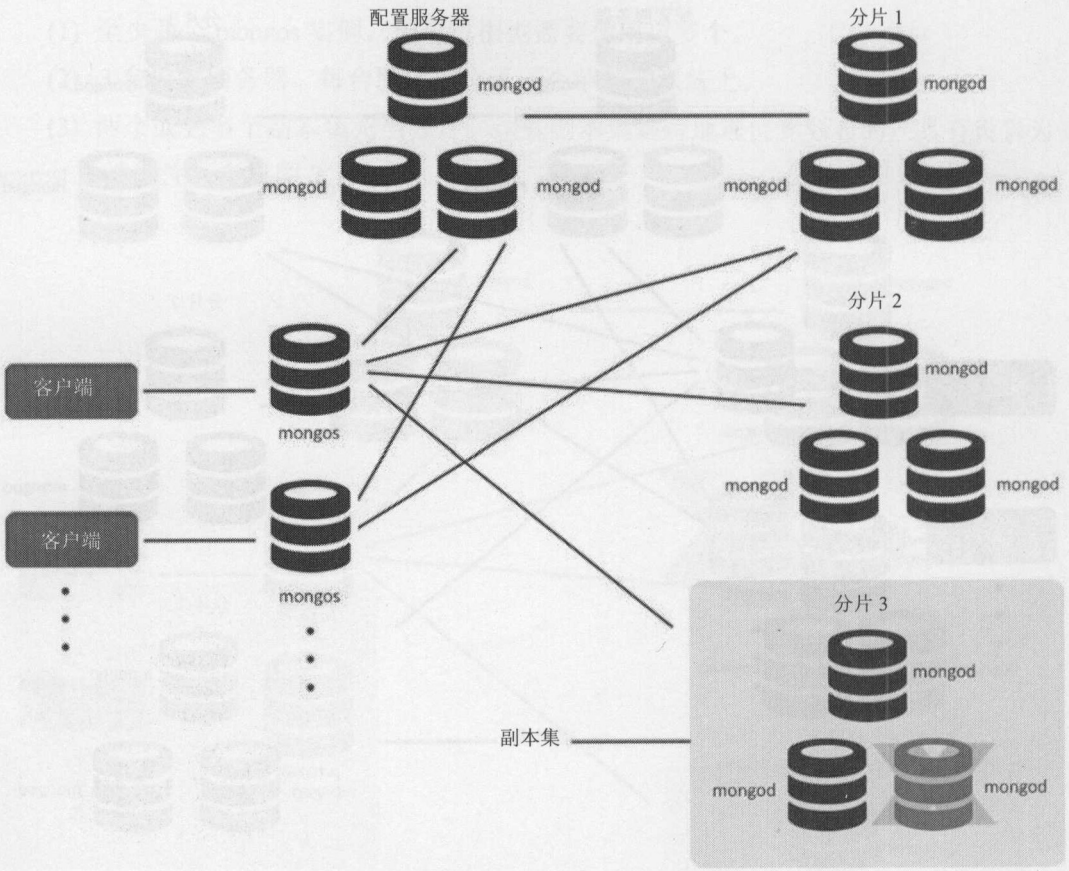

图 7-27 副本集的其中一个 mongod 不可用

唯一的区别在于，数据的复制会减少，从而让系统变得有点脆弱，因此你应该并行检查该 mongod 是否是可恢复的。如果是的话，则它应该恢复并且重启，而如果它是不可恢复的，那么需要创建一个新的副本集并且尽可能快地替换它。

7.6.3 场景 3

如果其中一个分片变得不可用：在此场景中，该分片上的数据将变得不可用，但其他分片将是可用的，因此这不会妨碍应用程序的运行。应用程序可以继续进行其读取/写入操作；不过，必须在应用程序内处理部分结果。同时，该分片应该试图尽可能快地恢复，参见图 7-28。

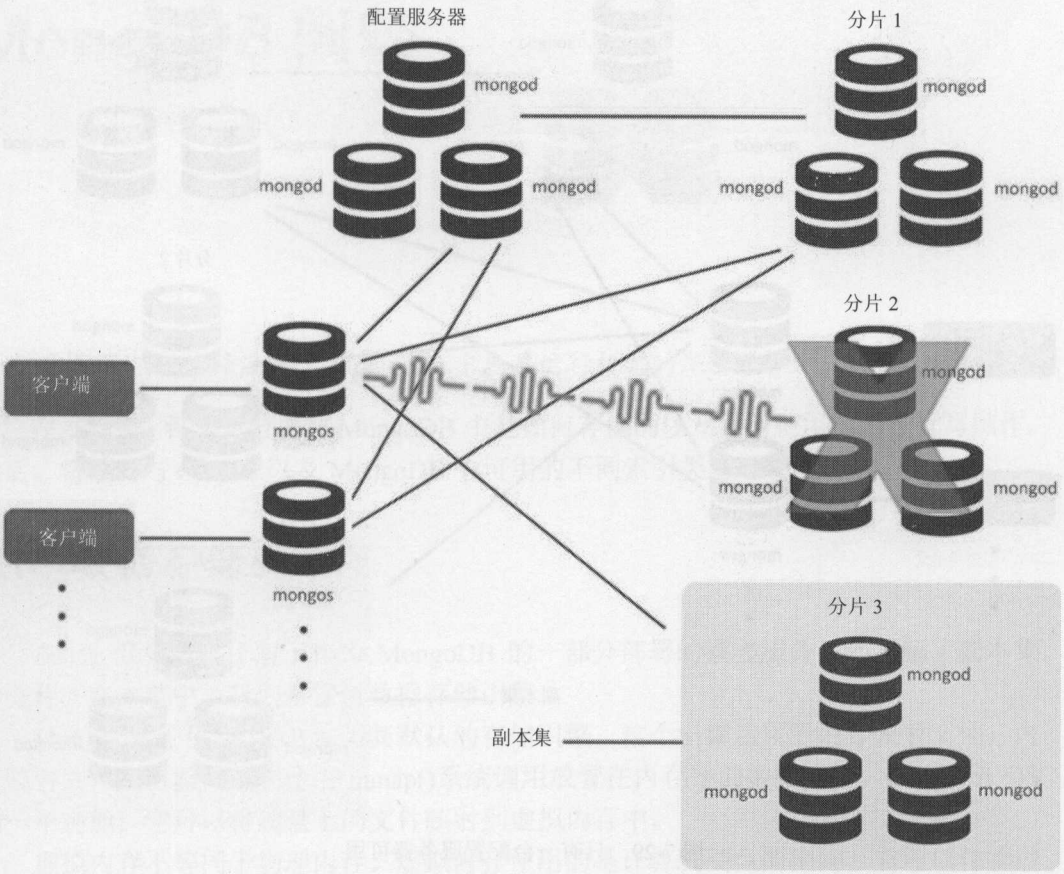

图 7-28 分片不可用

7.6.4 场景 4

3 台配置服务器中只有一台可用：在此场景中，尽管群集将变为只读，它将不会响应任何可能导致群集结构发生变化的操作，因而会导致元数据的变更，比如数据块迁移或者数据块划分。应该尽可能快地替换配置服务器，因为如果所有的配置服务器都变得不可用，那么将产生一个不可操作的群集，参见图 7-29。

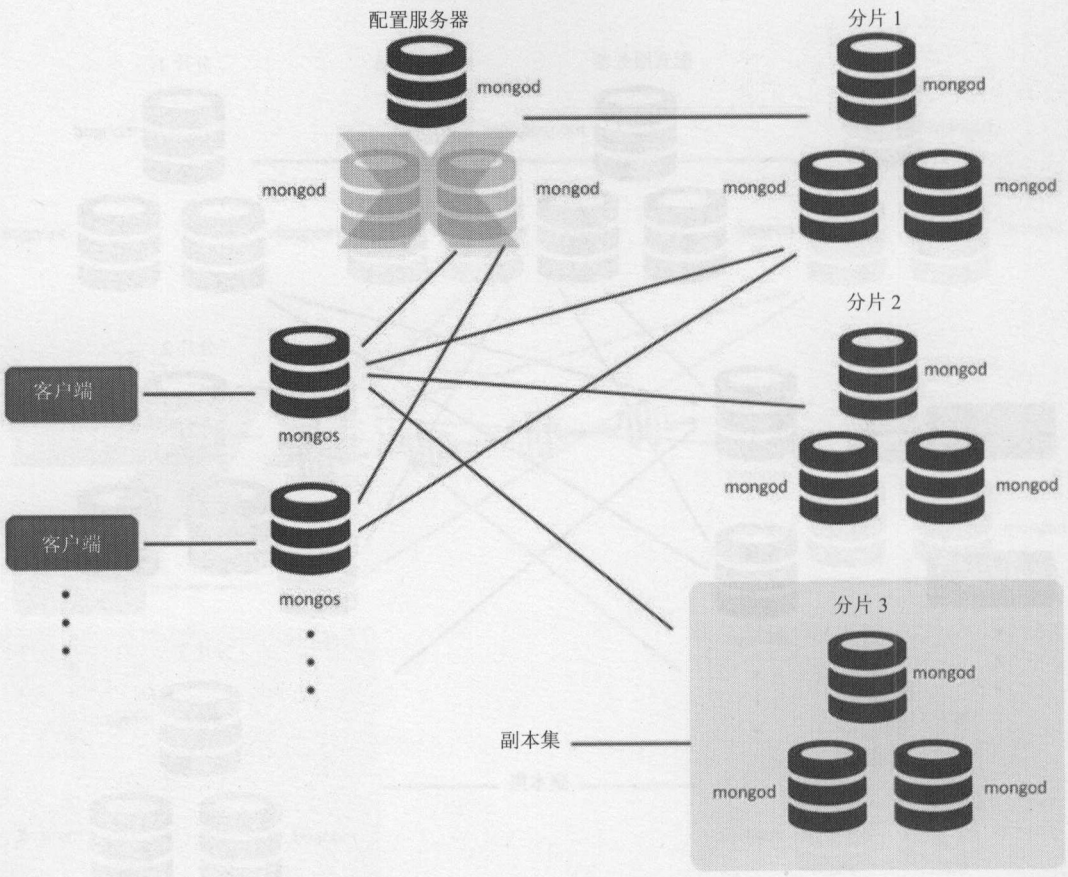

图 7-29 只有一台配置服务器可用

7.7 本章小结

在本章中，你了解了核心的程序和工具、独立部署、分片概念、复制概念以及生产环境部署。还了解了如何实现 HA(高可用性)。

在下一章中，将介绍在底层数据是如何被存储的、如何使用日志进行写操作、GridFS 的作用是什么，以及 MongoDB 中可用的不同的索引类型。

第 8 章

MongoDB 阐释

“MongoDB 阐释涵盖了 MongoDB 中底层运行机制的深层次概念。”

在本章中，将学习数据在 MongoDB 中是如何存储的以及如何使用日志进行写操作。最后，将学习与 GridFS 以及 MongoDB 中可用的不同索引类型有关的内容。

8.1 数据存储引擎

在上一章中，你了解了作为 MongoDB 的一部分部署的核心服务；还了解了副本集和分片。在本节中，我们将探讨数据存储引擎。

MongoDB 使用 MMAP 作为其默认的存储引擎。这个引擎适用于内存映射文件。内存映射文件是由操作系统使用 mmap()系统调用放置在内存中的数据文件。mmap 是 OS 的一个功能，它可以将硬盘上的文件映射到虚拟内存中。

虚拟内存不等同于物理内存。虚拟内存使用的是计算机硬盘的空间，它可以与物理 RAM 结合起来使用。

MongoDB 将内存映射文件用于所有的数据交互或者数据管理操作。当文档被访问时，这些数据文件就是被映射到内存的内存。MongoDB 允许 OS 控制内存映射并且分配 RAM 的最大数量。这样做就可以最小化 MongoDB 层面的工作量和编码量。这一缓存是基于 LRU 行为来实现的，其中最近最少用到的文件会被从工作集中移动到硬盘上，让空间尽可能地为新的最近和频繁使用的页面所用。

不过，这个方法有其自身的缺点。例如，MongoDB 无法控制将哪些数据存留在内存中以及从中移除哪些数据。因此每一次服务器重启都会导致页面错误，因为所访问的每个页面都将不再存在于工作集中，从而产生很长的数据检索时间。

MongoDB 也无法控制内存内容的优先级排序。对于特定的数据清理情形，它可以指明哪些内容需要在缓存中维护以及哪些需要被移除。例如，如果对一个没有索引的大型集合发起读取操作，则可能会导致将整个集合加载到内存的情形，这可能会造成 RAM 内容的清理，其中就包括移除掉可能非常重要的其他集合的索引。当任何位于 MongoDB 之外的外部程序试图访问很大一部分内存时，这一控制力的缺失可能还会造成分配给 MongoDB 的缓存减少；这最终将导致 MongoDB 响应的速度减慢。

在版本 3.0 发布之后，MongoDB 提供了一个可插拔的存储引擎 API，其中它使你可以根据工作量、应用程序需求以及可用的基础架构在存储引擎之间做出选择。

可插拔存储引擎层背后的愿景在于，让你可以使用一个数据模型、一种查询语言以及一组操作上的关注点，但在底层，存在许多为不同用例优化的存储引擎选项，如图 8-1 所示。

图 8-1 可插拔存储引擎 API

该可插拔存储引擎功能还为部署提供了灵活性，其中多种类型的存储引擎可以共存于相同的部署中。

MongoDB 版本 3.0 附带了两个存储引擎。

默认的 MMAPv1 是之前版本中使用的 MMAP 引擎的改进版本。更新后的 MongoDB MMAPv1 存储引擎实现了集合层面的并发控制。这个存储引擎擅长应对大量读取、插入以及就地更新的工作负荷。

新的 WiredTiger 存储引擎是由伯克利数据库(Berkeley DB)的架构师开发的，该数据库是世界上部署最广泛的嵌入式数据管理软件。WiredTiger 是基于现代多 CPU 架构的。它的目的是利用具有多核 CPU 和更多 RAM 的现代硬件的优势。

WiredTiger 会以压缩格式将数据存储在硬盘上。压缩可以将数据大小最多减少 70%(仅存储在硬盘上)以及将索引大小最多减少 50%(存储在硬盘和内存中都行)，这取决于所使用的压缩算法。除了减少存储空间之外，由于从硬盘读取的位数更少，因此压缩也能带来高得多的 I/O 可扩展性。它在较大的硬件利用率、较低的存储成本以及更高的可预见性能方面提供了显著的优势。

可以选择使用以下压缩算法：

- 默认的是 Snappy，它被用于文档和日志。它提供了很好的压缩率，而这只需要很少的 CPU 开销。根据数据类型的不同，其压缩率大约在 70%上下。
- zlib 提供了非常棒的压缩，但需要额外的 CPU 开销。

- 前缀压缩被默认用于索引，以便将索引存储的内存占用空间降低约 50%(取决于工作负荷)以及释放更多的工作集用于频繁访问的文档。

管理员可以为所有集合以及索引修改默认的压缩设置。也可以在集合与索引创建期间，为每个集合与每个索引配置不同的压缩。

WiredTiger 还提供了文档级别粒度的并发性。写操作不会再被其他写操作阻塞，除非它们都是在访问相同的文档。因此它支持读取者和写入者对集合中文档的并发访问。客户端可以在写操作处理过程中读取文档，而同时多线程可以修改一个集合中的不同文档。因此它擅长应对写入密集型的工作负荷(写入性能提升 7～10 倍)。

较高的并发性也能驱动基础架构的简化。应用程序可以完全利用可用的服务器资源，从而简化需要用来满足性能 SLA(服务水平协议)的架构。在使用 MongoDB 早前版本的更为粗粒度的数据库级别的锁定时，即便是在主机系统仍然有充足内存、I/O 带宽以及硬盘容量可用的情况下，用户通常也必须实现分片，以便收缩由对数据库的单个写入锁引发延迟而拖缓处理的工作负荷。细粒度的并发性会降低系统开销从而获得更大的系统利用率，从而避免不必要的开销和管理工作量。

这个存储引擎为你提供了基于集合和索引级别的控制，以决定压缩以及不压缩的内容。

WiredTiger 存储引擎仅可用于 64 位的 MongoDB。

WiredTiger 通过其缓存来管理数据。WiredTiger 存储引擎提供了更多的内存控制，让你可以配置为 WiredTiger 缓存分配多少 RAM，这个值默认为 1GB 或者可用内存的 50%，而无论哪一个都显得过于大了些。

接下来将简要了解如何将数据存储到硬盘上。

8.2 (与 MMAPv1 相关的)数据文件

首先，我们来查看数据文件。如你所见，在核心服务中，mongod 使用的默认数据目录是/data/db/。

这个目录中有用于每一个数据库的单独文件。每一个数据库都有一个.ns 文件以及具有单向递增的数值扩展名的多个数据文件。

例如，如果创建一个名称为 mydbpoc 的数据库，那么它将被存储到以下文件中：mydb.ns、mydb.1、mydb.2 等，如图 8-2 所示。

对于数据库的每一个新的数值数据文件，其大小将是其前一个数值数据文件大小的两倍。文件大小的限制是 2GB。如果该文件大小达到 2GB，那么所有后续的数字编号文件也将保持 2GB 大小。这是一个故意如此设计的行为。这一行为会确保小数据库不会浪费过多的硬盘空间，而大数据库主要保存在硬盘上的连续区域。

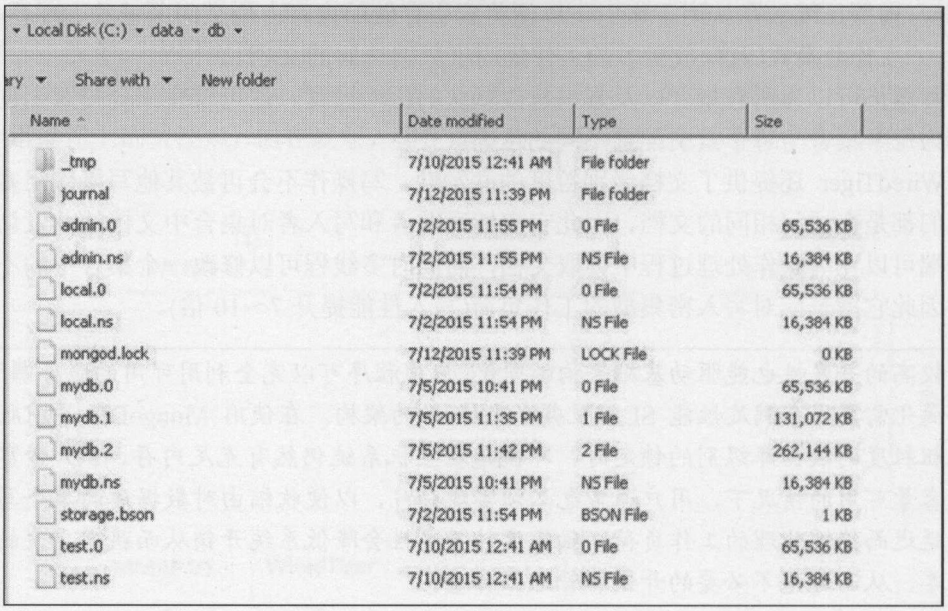

图 8-2 数据文件

注意，为了确保一致的性能，MongoDB 会预先分配数据文件。预先分配是在后台进行的，并且是在每次填充一个数据文件时启动的。这意味着 MongoDB 服务器总是会尝试为每一个数据库保留一个额外的空数据文件，以避免文件分配时出现阻塞。

如果有多个小的数据库存在于硬盘上，那么使用 storage.mmapv1.smallFiles 选项将降低这些文件的大小。

接下来，将看到在底层是如何实际存储数据的。双链表是用于存储数据的关键数据结构。

命名空间(.ns 文件)

在数据文件中，你的数据空间会被划分成多个命名空间，其中命名空间对应于一个集合或者一个索引。

这些命名空间的元数据存储在.ns 文件中。如果检查数据目录，将发现一个名称为[dbname].ns 的文件。

用于存储元数据的.ns 文件的大小是 16MB。这个文件可以被看作一个大的哈希表，它被划分成小的存储桶，其大小约为 1KB。每个存储桶会存储特定于一个命名空间的元数据，参见图 8-3。

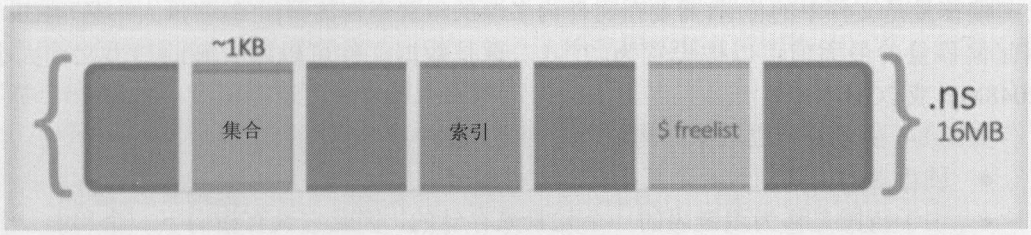

图 8-3 命名空间数据结构

1. 集合命名空间

如图 8-4 所示，集合命名空间存储桶包含了一些元数据，如下所示：

- 集合的名称
- 集合上的一些统计信息，比如计数、大小等。(这就是无论何时对集合进行计数时它会返回快速响应的原因。)
- 索引详细信息，因此它可以维护到所创建的每一个索引的链接
- 一个删除列表
- 存储范围详情的一个双链表(它会存储到第一个和最后一个范围信息的指针)

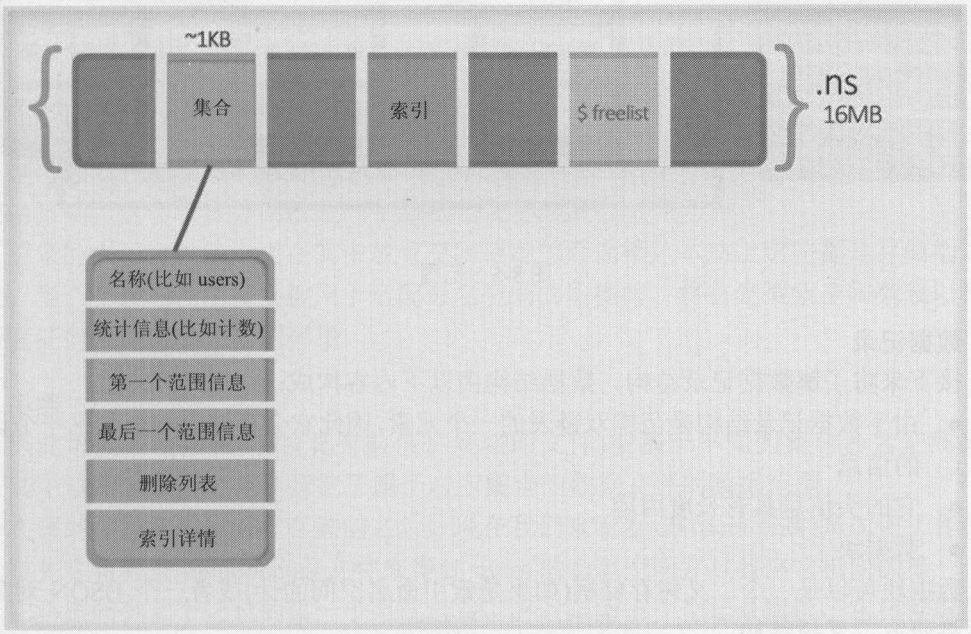

图 8-4 集合命名空间详细信息

范围

范围指的是一个数据文件中的一个数据记录分组，因此一个范围分组就组成了一个命名空间的完整数据。范围使用硬盘位置来指向实际驻留数据的硬盘位置。它由两部分构成：文件编号和偏移量。

文件编号指定了它指向的数据文件(0、1 等)。

偏移量是文件中的位置(需要在文件内多深的位置查询数据)。偏移量大小是 4KB。因此偏移量的最大值可以被设置为 2^{31}-1，这是数据文件可以增长到的最大文件大小(2048MB 或 2GB)。

如图 8-5 所示，范围数据结构由以下内容构成：

- 硬盘上的位置，这就是其指向的文件编号。
- 由于范围是作为双链表的一个元素来存储的，因此它具有指向下一个和上一个范围的指针。
- 一旦它具有了其指向的文件编号，其指向的文件中的数据记录分组就会被进一步存储为双链表。因此它会维护到其指向数据块的第一个数据记录和最后一个数据记录的指针，也就是文件中的偏移量(数据存储在文件多深的位置)。

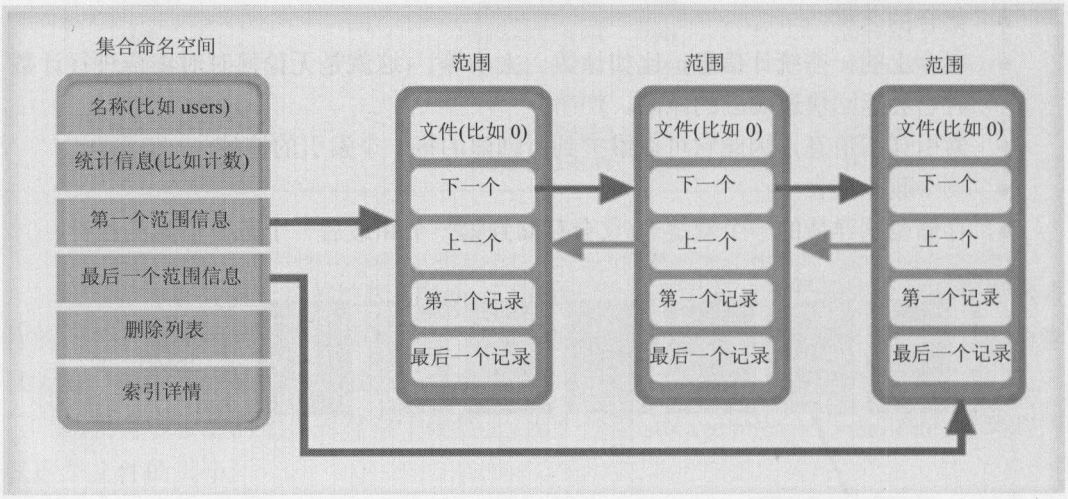

图 8-5 范围

数据记录

接下来将了解数据记录结构。数据结构由以下内容构成：

- 由于数据记录结构是范围双链表的一个元素，因此它会存储上一个和下一个记录的信息
- 它的头信息具有长度内容
- 数据块

数据块可以是一个二叉树存储桶(如果是索引命名空间的话)或者一个 BSON 对象。稍后将研究二叉树结构。

BSON 对象对应于集合的实际数据。BSON 对象的大小无需与数据块的大小相同。默认使用了 2 的幂次大小这一配置，这样每一个文档就会被存储在一个包含文档外加额外空间或空白区域的空间中。这一设计决策很有用，可以在更新操作导致对象大小发生变化时避免出现一个对象从一个块移动到另一个块的情况。

MongoDB 支持多分配策略，这可以确定如何将空白区域添加到一个文档，参见图 8-6。由于就地更新比重新定位更有效率，因此所有的空白区域策略都会用额外的空间换

取更高的效率以及更少的碎片。不同的策略支持不同的工作负荷。例如，正好合适的分配适用于仅有插入工作负荷的集合，其中集合大小是固定且永远不变的，而 2 的幂次配置对于插入/更新/删除这些工作负荷很有效。

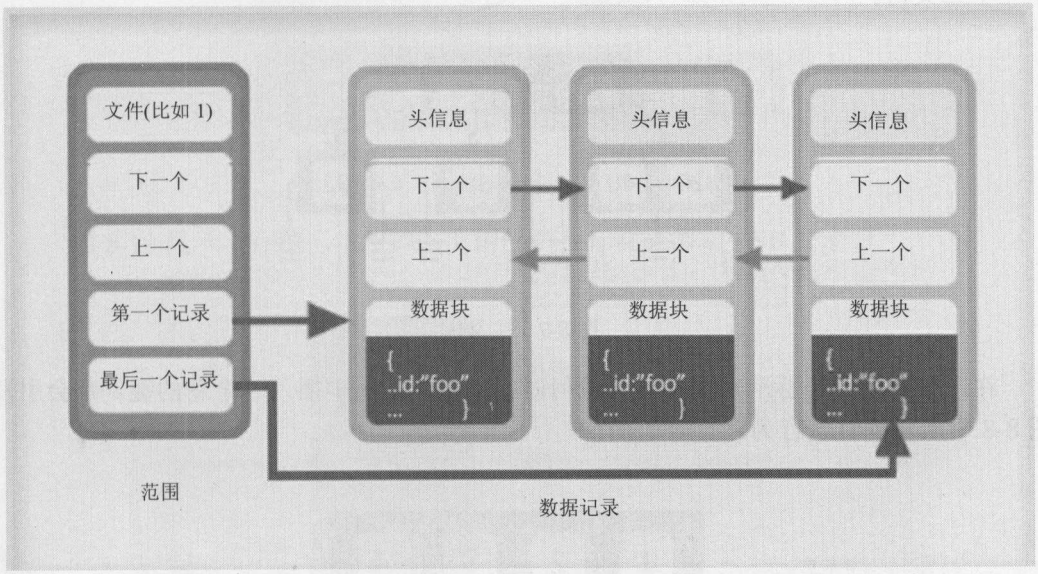

图 8-6 记录数据结构

删除列表

删除列表存储了数据被删除或移除的范围的详细信息(无论更新何时造成大小变更出现的移动，这会导致分配的空间中数据出现不匹配)。

记录的大小可以确定要在其中放置空闲范围的存储桶。本质上这些都是存储桶式单链表。当一个新的范围需要被用于适配命名空间的数据时，将首先搜索空闲列表以检查是否有任何合适大小的范围可用。

小结

因此可以假定数据文件(具有数值扩展名的文件)被基于不同的集合命名空间来划分，其中该命名空间的范围指定了属于对应集合的数据文件的数据范围。

在理解了数据是如何被存储的之后，现在我们来看看 db.users.find()是如何工作的。

首先它将检查 mydbpoc.ns 文件来得到 users 的命名空间并且找出它指向的第一个范围。它将跟随到首个记录的第一个范围链接，并且跟随下一个记录指针，它将读取第一个范围的数据记录直到得到最后一个记录。然后它将以类似方式跟随下一个范围指针以读取其数据记录。在读取最后一个范围数据记录之前，它都会遵循这一模式。

2. $freelist

.ns 文件有一个用于范围的被称为$freelist 的特殊命名空间。$freelist 会记录不再被使用的范围，比如被丢弃的索引或集合的范围。

3. 索引二叉树

现在我们来看看索引是如何被存储的。二叉树结构被用于存储索引。图 8-7 中显示了一个二叉树。

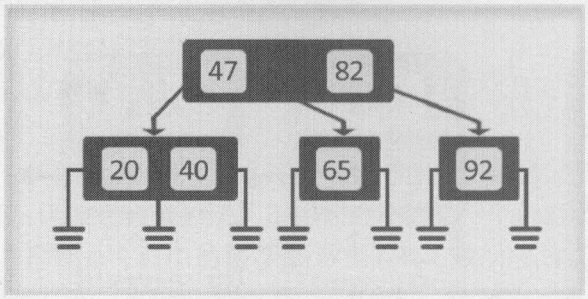

图 8-7 二叉树

在二叉树的一个标准实现中，无论何时在一个二叉树中插入一个新的键，都会引发图 8-8 中所示的默认行为。

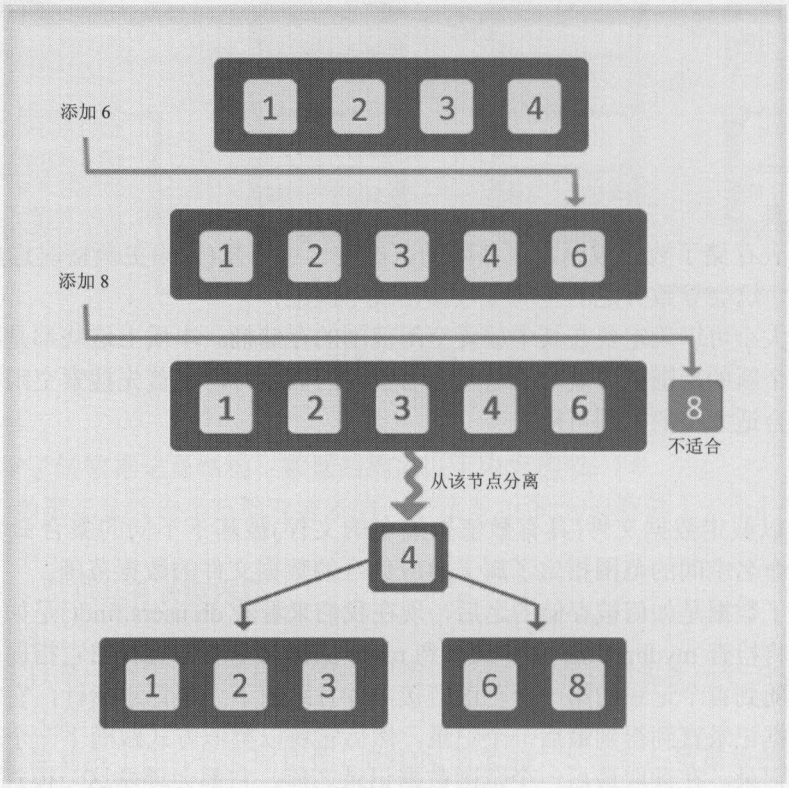

图 8-8 二叉树标准实现

MongoDB 实现二叉树的方式稍微有点不同。

在上述场景中，如果具有像时间戳、ObjectID 或者一个递增数值这样的键，那么存储桶将总是半满状态，这会导致大量的空间浪费。

为了克服这个问题，MongoDB 对其稍微进行了一些修改。无论它何时识别出索引键

是一个递增键，那么它就不会进行 50/50 的划分，而是进行如图 8-9 所示的 90/10 的划分。

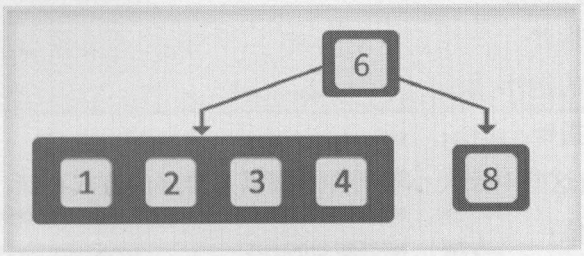

图 8-9　MongoDB 的二叉树 90/10 划分

图 8-10 显示了存储桶结构。二叉树的每个存储桶都是 8KB 大小。

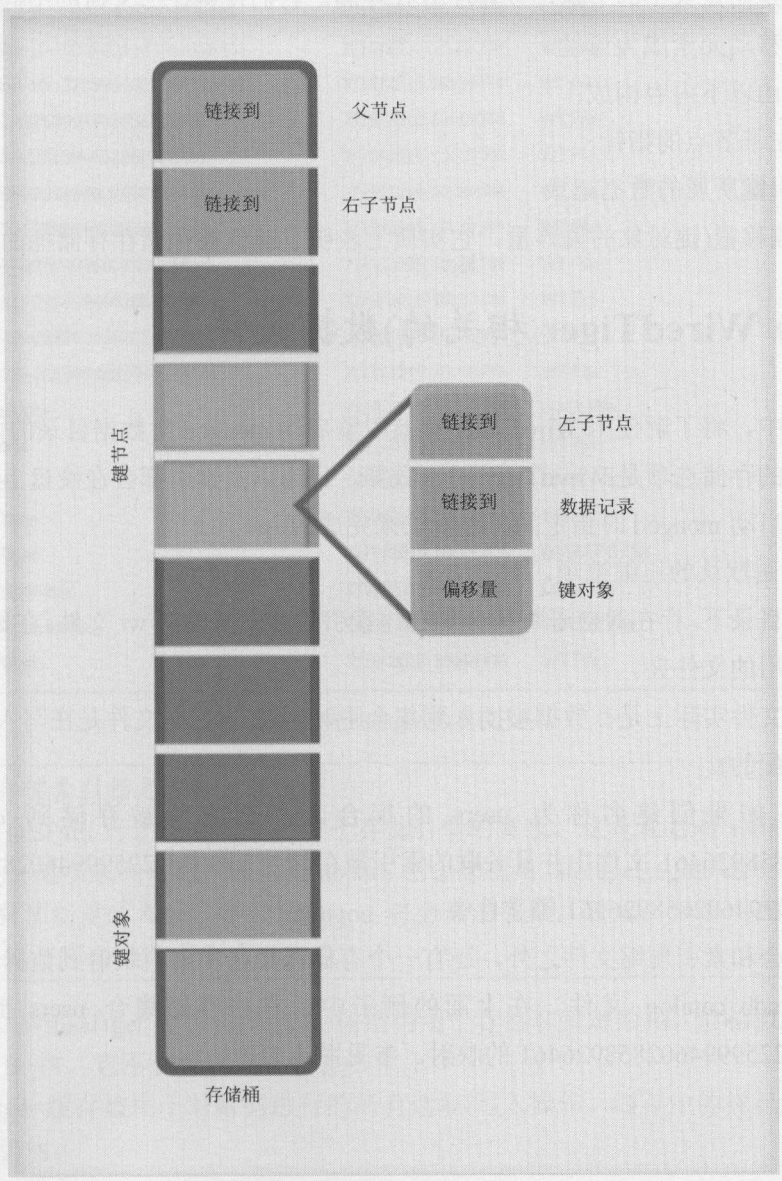

图 8-10　二叉树存储桶数据结构

存储桶由以下内容构成：

- 到父节点的指针
- 到右子节点的指针
- 到键节点的指针
- 一组键对象(这些对象大小不同并且都以未排序的方式存储；这些对象实际上就是索引键的值)

键节点

键节点是固定大小的节点，以排序方式存储。它们使得在二叉树的不同节点之间轻易地划分和移动元素成为可能。

键节点由以下内容构成：

- 到左子节点的指针
- 索引键所属的数据记录
- 键偏移量(键对象的偏移量，它实质上表明了键值被存储在存储桶的什么位置)

8.3 (与 WiredTiger 相关的)数据文件

在本节中，将了解使用 WiredTiger 存储引擎启动 mongod 时数据目录的内容。

当选择的存储选项是 WiredTiger 时，数据、日志以及索引都会在硬盘上被压缩。该压缩是基于启动 mongod 时指定的压缩算法来完成的。

Snappy 是默认的压缩选项。

在数据目录下，存在对应于每一个集合与索引的独立压缩的 wt 文件。在数据目录中，日志有其专用的文件夹。

压缩的文件实际上是在数据被插入到集合中时创建的(这些文件是在写入时分配的，而非预先分配的)。

例如，如果创建名称为 users 的集合，那么它将被存储到 collection-0-2259994602858926461 文件中并且关联的索引被存储到 index-1-2259994602858926461、index-2-2259994602858926461 等文件中。

除了集合和索引压缩文件之外，还有一个存储将集合和索引映射到数据目录文件的元数据的 mdb_catalog 文件。在上面的例子中，它将存储集合 users 到 wt 文件 collection-0-2259994602858926461 的映射。参见图 8-11。

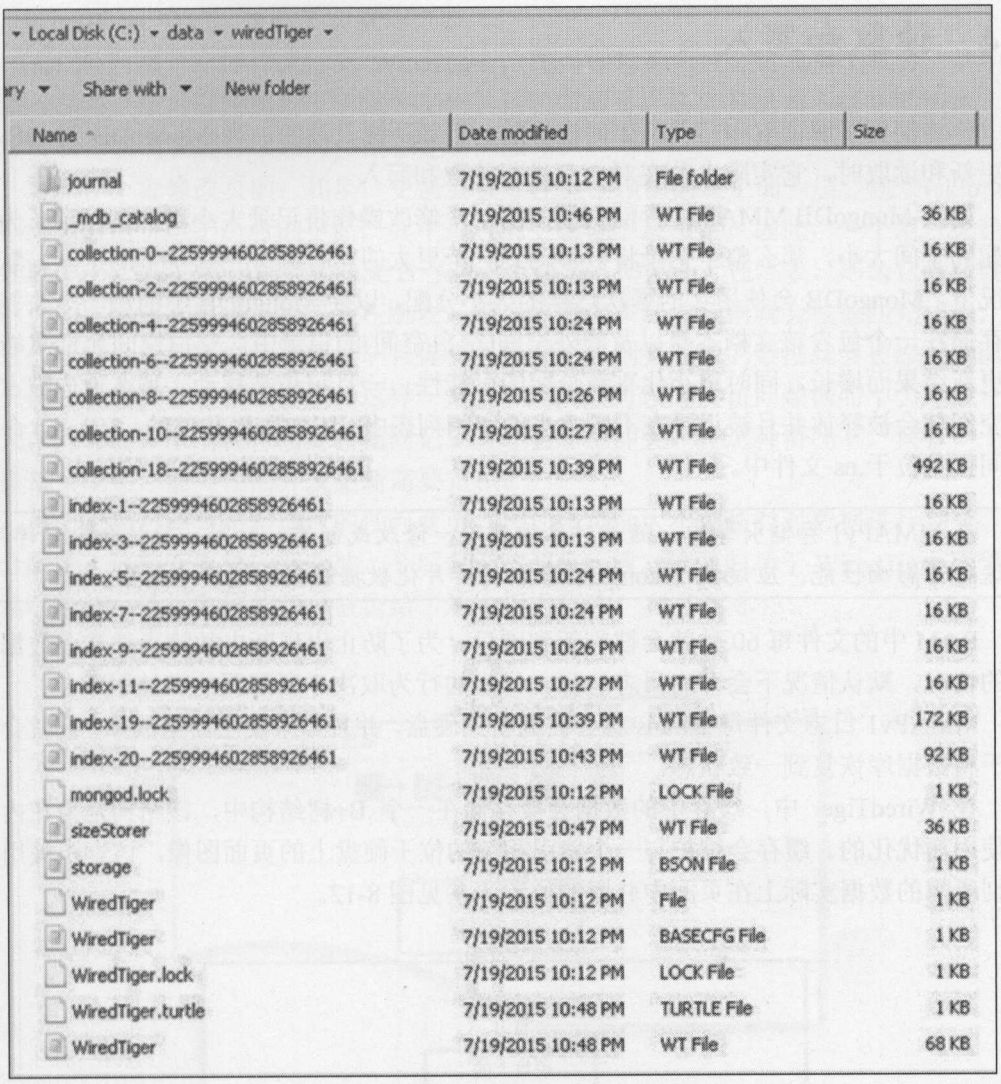

图 8-11 WiredTiger 数据文件夹

可以为存储索引指定单独的容量。

在指定 DBPath 时，需要确保对应于存储引擎的目录，这是在启动 mongod 时使用–storageEngine 选项来指定的。如果该 dbpath 包含除使用–storageEngine 选项指定的文件之外由存储引擎创建的文件，那么 mongod 将会启动失败。因此，如果 DBPath 中存在 MMAPv1 文件，那么 WT 将会启动失败。

在内部，WiredTiger 会将传统的 B+树结构用于存储和管理数据，但相似之处仅此而已。不同于 B+树，它不支持就地更新。

WiredTiger 缓存被用于对数据进行的所有读取/写入操作。缓存中的树已经被优化以便用于内存访问。

8.4 读取和写入

你将简要了解读取和写入是如何进行的。正如所提及过的，当 MongoDB 对 DB 进行更新和读取时，它实际上是在对内存进行读取和写入。

如果 MongoDB MMAPv1 存储引擎中的一个修改操作将记录大小增加到超出了为其分配的空间大小，那么整个记录将被移动到一个更大的具有额外空白字节的空间。默认情况下，MongoDB 会使用 2 的幂次这样的大小分配，以便 MongoDB 中的每一个文档都被存储在一个包含该文档本身以及额外空间(空白空间)的记录中。空白空间允许文档随着更新结果而增长，同时最小化重新分配的可能性。一旦记录被移动，那么原先被占用的空间就会被释放并且被记录在不同大小的空闲列表中。正如所提及过的，$freelist 命名空间正是位于.ns 文件中。

在 MMAPv1 存储引擎中，随着对象被删除、修改或创建，时间一长就会出现碎片，而这将会影响性能。应该执行 compact 命令将碎片化数据移动到连续空间中。

RAM 中的文件每 60 秒就会被刷新到硬盘。为了防止出现断电故障事件造成数据丢失的情况，默认情况下会开启日志运行。日志的行为取决于所配置的存储引擎。

MMAPv1 日志文件每 100ms 就会被刷新到硬盘，并且如果发生断电故障，它就会被用于将数据库恢复到一致状态。

在 WiredTiger 中，缓存中的数据会被存储在一个 B+树结构中，该结构是为在内存中使用而优化的。缓存会维护与一个索引相关的位于硬盘上的页面图像，该索引被用于识别所需的数据实际上在页面中驻留的位置，参见图 8-12。

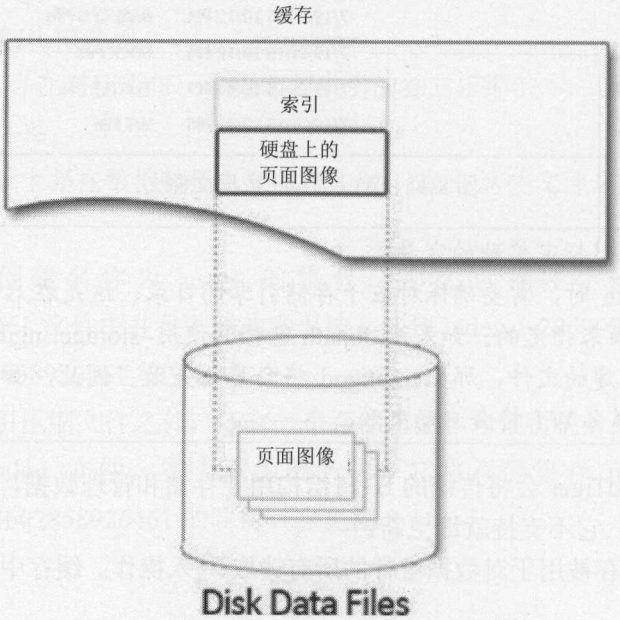

图 8-12 WiredTiger 缓存

WiredTiger 中的写操作不会就地更新。

无论何时将一个操作发送到 WiredTiger，在内部它都会被分解成多个事务，其中每个事务都在一个内存快照的上下文中工作。该快照就是在事务开始前所提交的版本。写入者可以在读取者进行读取时并发创建新的版本。

写操作不会修改页面；相反，更新会被放在页面顶部的最上面一层。跳表(skipList)数据结构被用于维护所有的更新，其中最近的更新位于最顶层。因此，一个用户无论何时读取/写入数据，索引都会检查是否存在一个跳表。如果不存在跳表，那么它会从硬盘上的页面图像返回数据。如果存在跳表，那么该跳表头部的数据就会被返回到线程，然后这些线程会对数据进行更新。一旦提交执行完毕，则更新后的数据就会被添加到跳表的头部，且指针会被相应调整。这样一来多个用户就可以并发访问数据而不会造成任何冲突。只有在多个线程试图更新相同记录时才会产生冲突。在这种情况下，只有一个更新操作会成功，而其他的并发更新需要重试。

由于更新对该树结构造成的任何变更，比如当页面大小增长时对数据进行划分、重新分配等，这些变更稍后都会被一个后台程序调整一致。这要归功于 WiredTiger 引擎的快速写操作；数据整理的任务被留给了这个后台程序。参见图 8-13。

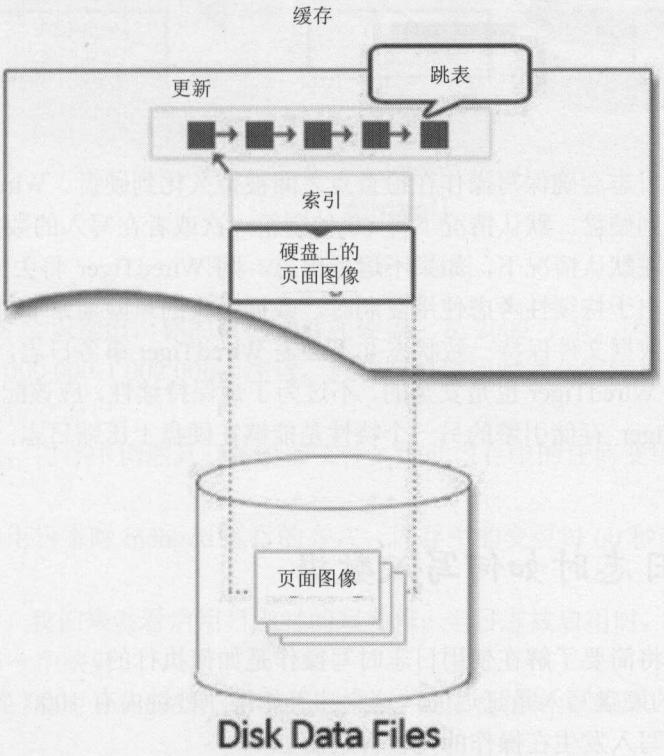

图 8-13 跳表

WiredTiger 使用了 MVCC 方式来确保以并发控制的方式来维护多版本的数据。该方式还确保了每个尝试访问数据的线程都会看到数据的最一致版本。如你所见，写操作并

非就地方式的；相反，它们会被附加到顶部具有最近更新的跳表数据结构中数据的顶部。访问数据的线程会得到最近的副本，并且它们将不中断地继续使用该副本，直到它们进行提交。一旦它们进行提交，其更新就会被附加到跳表的顶部，此后所有访问该数据的线程都会看到最新的更新。

这使得多线程可以并发地访问相同数据，而不会出现任何锁或竞争。这还使得写入者可以在读取者进行读取时并发创建新的版本。只有在多个线程试图更新相同记录时才会产生冲突。在这种情况下，只有一个更新操作会成功，而其他的并发更新需要重试。

图 8-14 描述了运行中的 MVCC。

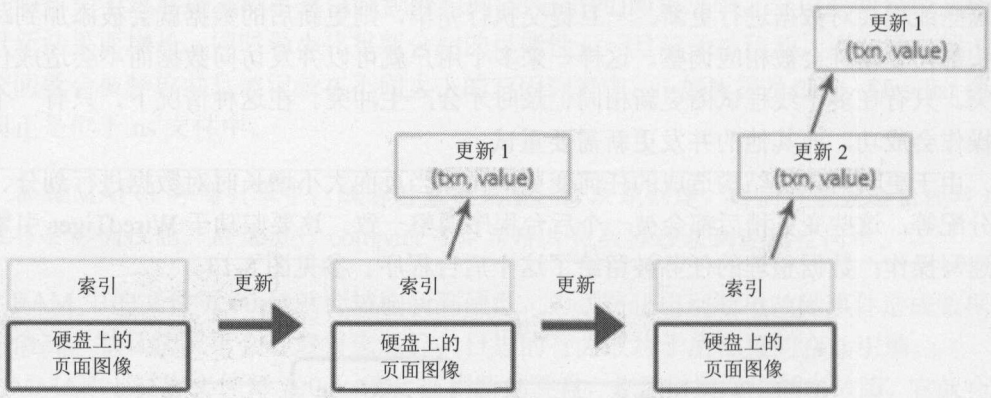

图 8-14　运行中的更新

WiredTiger 日志会确保写操作在检查点之间被持久化到硬盘。WiredTiger 使用检查点来将数据刷新到硬盘，默认情况下每 60 秒刷新一次或者在写入的数据超过 2GB 时进行刷新。因此，在默认情况下，如果不运行日志，则 WiredTiger 将丢失至多 60 秒内的写操作，不过在出于持续性考虑使用复制时，数据丢失的风险通常非常小。在非正常关机的事件中，将数据文件保持一致状态并不必需 WiredTiger 事务日志，因此在不启用日志的情况下运行 WiredTiger 也是安全的，不过为了确保持续性，应该配置“副本安全的”写关注。WiredTiger 存储引擎的另一个特性是能够在硬盘上压缩日志，因此会减少存储空间。

8.5　使用日志时如何写入数据

在本节中，将简要了解在使用日志时写操作是如何执行的。

MongoDB 的硬盘写入是延迟的，这意味着如果一秒钟内有 1000 个增长，它也只会写入一次。物理写入发生在操作的几秒钟之后。

现在将看到 mongod 中更新实际上是如何进行的。

如你所知，在 MongoDB 系统中，mongod 是主要的守护程序。因此硬盘存有数据文件和日志文件。参见图 8-15。

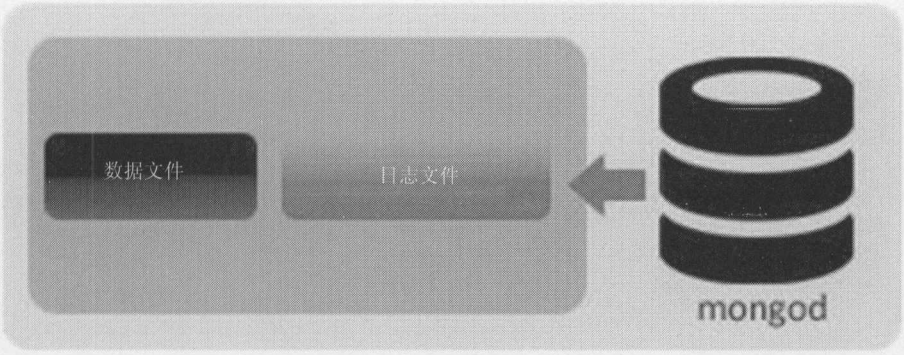

图 8-15 mongod

当 mongod 启动时，数据文件会被映射到一个分片视图。也就是说，数据文件会被映射到一个虚拟的地址空间。参见图 8-16。

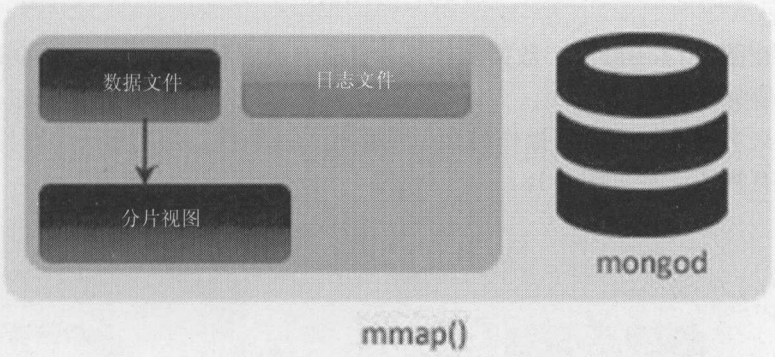

图 8-16 映射到分片视图

实质上，OS 会识别出，你的数据文件在硬盘上是 2000 字节，因此它会将该文件映射到内存地址 1 000 000-1 002 000。注意，只有在被访问时才会实际加载该数据；OS 只是映射它并且保留它而已。

到目前为止，内存中仍然具有备份的文件。因此内存中的任何变更都会被 OS 刷新到底层文件。

这就是不启用日志时 mongod 运行的方式。内存中的变更每 60 秒就会被 OS 刷新到硬盘。

在此场景中，我们来看看启用日志时的写操作。当日志被启用时，mongod 会对一个私有视图进行另一个映射。

这就是启用日志时 mongod 所使用的虚拟内存数量会倍增的原因。参见图 8-17。

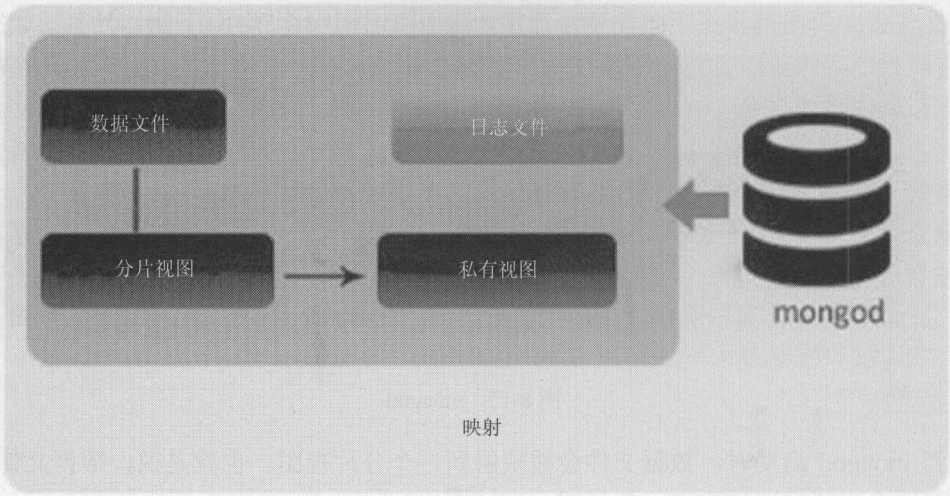

图 8-17　映射到私有视图

可以在图 8-17 中看到，数据文件并非直接连接到私有视图，因此 OS 不会将变更从私有视图刷新到硬盘。

我们来看看发起一个写操作时会引发哪些系列事件。当发起一个写操作时，首先会写入到私有视图，参见图 8-18。

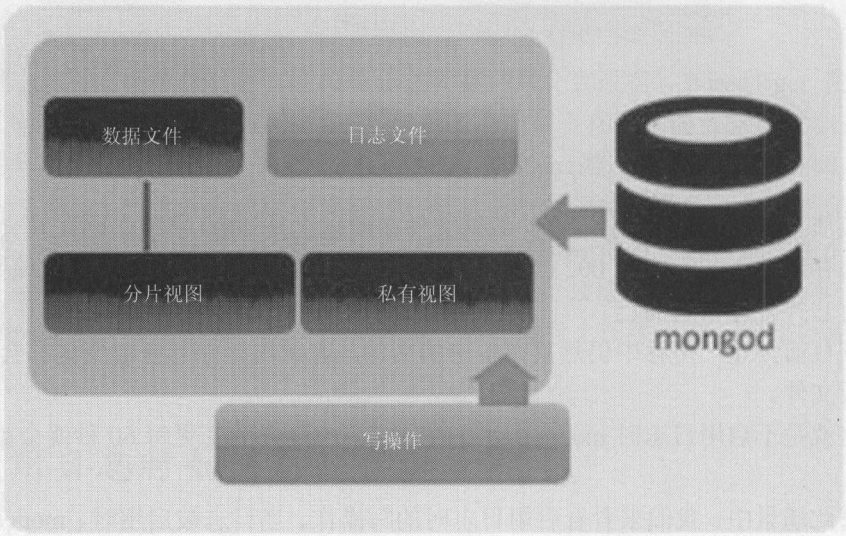

图 8-18　发起写操作

接下来，变更会被写入到日志文件，附加文件中修改了什么的简要描述，参见图 8-19。

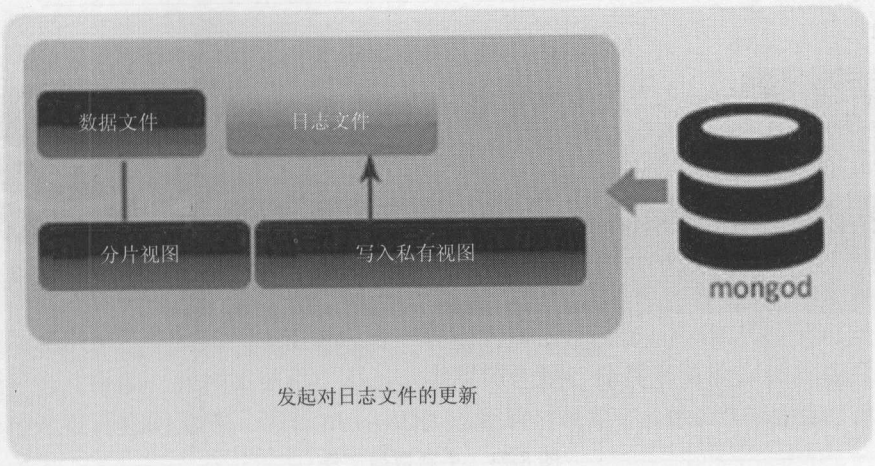

图 8-19 更新日志文件

当日志得到变更时，它就会持续附加变更描述。如果此时 mongod 出现故障，那么即使数据文件还没有被修改，日志也会重复所有的变更，以便让此时的写操作安全。

现在日志将重复该分片视图上记录的变更，参见图 8-20。

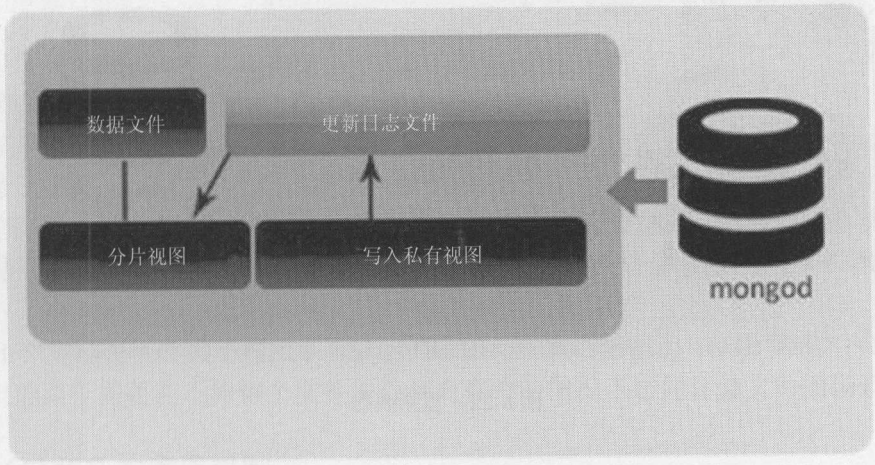

图 8-20 更新分片视图

最后，会用非常快的速度将变更写到硬盘。默认情况下，mongod 将每 60 秒请求 OS 这样做一次，参见图 8-21。

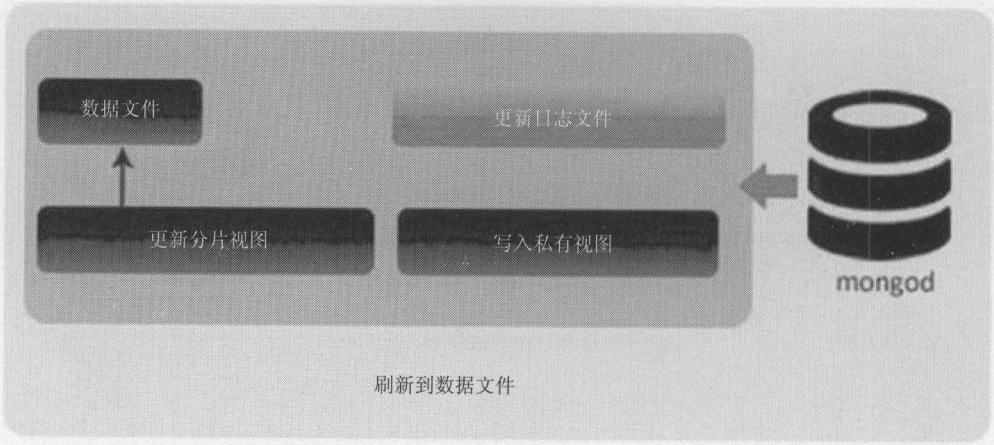

图 8-21　更新数据文件

在最后一步中，mongod 将分片视图重新映射到了私有视图。这样做是为了防止私有视图变得过于脏，参见图 8-22。

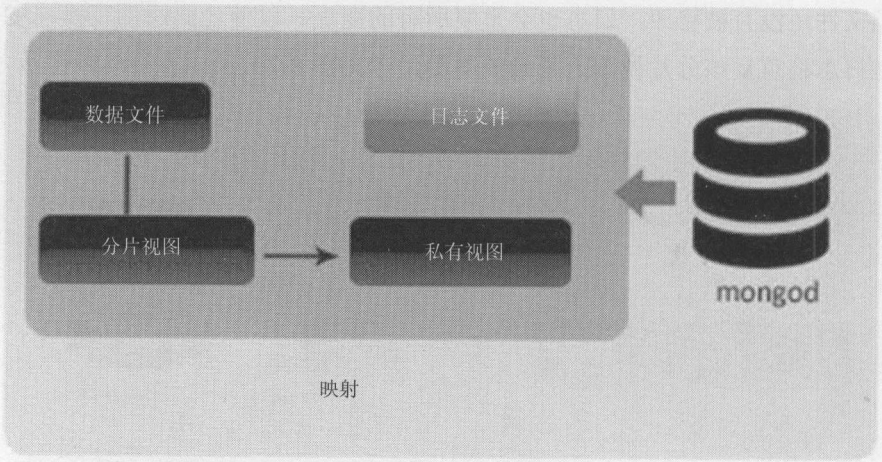

图 8-22　重新映射

8.6　GridFS——MongoDB 文件系统

你了解了底层发生了什么。你看到了 MongoDB 将数据存储在 BSON 文档中。BSON 文档具有 16MB 这一文档大小限制。

GridFS 是 MongoDB 用于处理超出 BSON 文档大小限制的大型文件的规范。本节将简要介绍 GridFS。

此处的“规范”一词意味着它本身并非 MongoDB 的一个特性，因此 MongoDB 中并没有实现它的代码。它仅指定了多大的文件需要被处理。像 PHP、Python 等这样的语言驱动程序实现了这一规范并且对使用该驱动程序的用户公开了 API，让他们可以在 MongoDB 中存储/检索大型文件。

8.6.1 GridFS 的基本原理

根据设计，MongoDB 的文档(比如一个 BSON 对象)不能大于 16MB。这是为了将性能保持在最佳水平，并且其大小正好适合于我们的需求。例如，4MB 的空间可能就足以存储一段声音片段或一张档案图片。不过，如果需求是存储高质量的音频或视频片段、甚至大小超过数百兆的文件，那么 MongoDB 已经通过使用 GridFS 为你提供了服务。

GridFS 为跨多个文档划分一个大型文件指定了一种机制。实现该机制的语言驱动程序，比如 PHP 驱动程序，将负责在底层划分所存储的文件(或者在检索文件时合并划分的数据块)。使用该驱动程序的开发人员不需要知道其内部细节。这样一来，GridFS 就使得开发人员可以用一种透明和有效的方式来存储和操作文件。

GridFS 使用了两个集合用于存储文件。一个集合维护文件的元数据，而另一个集合通过将文件数据分解成称为数据块的小块来存储文件的数据。这意味着该文件会被划分成较小的数据块，且每一个数据块会被存储为一个独立的文档。默认情况下，数据块的大小被限制为 255KB。

此方式不仅让数据存储变得可扩展且易于实现，还让范围查询在检索文件的特定部分时更易于使用。

无论何时在 GridFS 中查询一个文件，都会按照客户端的要求重新装配数据块。这也为用户提供了访问文件任意部分的能力。例如，用户可以直接移动到一个视频文件的中间。

GridFS 规范在文件大小超出了 MongoDB BSON 文档默认的 16MB 限制时会非常有用。它也能用于需要在不将整个文件加载到内存中的情况下访问存储文件的场景。

8.6.2 GridFS 的底层机制

GridFS 是用于存储文件的一个轻量级规范。

对于 GridFS 请求来说，并没有在 MongoDB 服务器上完成处理的“特殊情况”。所有的工作都是在客户端完成的。

GridFS 可以通过将大型文件划分成较小的数据块并且将每个数据块存储为独立文档来存储大型文件。除了这些数据块之外，还有一个文档包含了关于文件的元数据。使用这些元数据信息，数据块就会被组合到一起，形成完整的文件。

用于数据块的存储开销可以保持到最小化，因为 MongoDB 支持在文档中存储二进

制数据。

GridFS 用于存储大型文件的两个集合，其默认名称为 fs.files 和 fs.chunks，不过相较于 fs，也可以选择不同的存储桶名称。

默认情况下，数据块会被存储到 fs.chunks 集合中。如果需要，这可以被重写。因此所有的数据都包含在 fs.chunks 集合中。

数据块集合中独立文档的结构非常简单：

```
{
"_id" :ObjectId("..."),"n" : 0,"data" : BinData("..."),
"files_id" :ObjectId("...")
}
```

数据块文档具有以下重要的键。

- **“_id”**：这是唯一标识符。
- **“files _id”**：这是包含与数据块相关的元数据的文档的唯一标识符。
- **“n”**：这主要是描述数据块在原始文件中的位置。
- **“data”**：这是构成这一数据块的实际二进制数据。

fs.files 集合会为每一个文件存储元数据。这个集合中的每个文档都代表着 GridFS 中的单个文件。除了通用的元数据信息之外，该集合的每一个文档也可以包含特定于它所代表的文件的自定义元数据。

下面是由 GridFS 规范托管的键：

- **_id**：这是文件的唯一标识符。
- **Length**：这描述了组成文件完整内容的字节数合计值。
- **chunkSize**：这是文件的数据块大小，以字节计。默认情况下，其值为 255KB，但如果需要的话，可以调整它。
- **uploadDate**：这是文件被存储在 GridFS 中时的时间戳。
- **md5**：这是在服务器端生成的，并且是文件内容的 md5 校验和。MongoDB 服务器会通过使用 filemd5 命令来生成其值，该命令会计算上传数据块的 md5 校验和。这表明，用户可以检查这个值来确保文件被正确上传。

一个典型的 fs.files 文档看起来如下所示，也可以参见图 8-23：

```
{
  "_id" :ObjectId("..."), "length" : data_number,
  "chunkSize" : data_number,
  "uploadDate" : data_date,
  "md5" :data_string
}
```

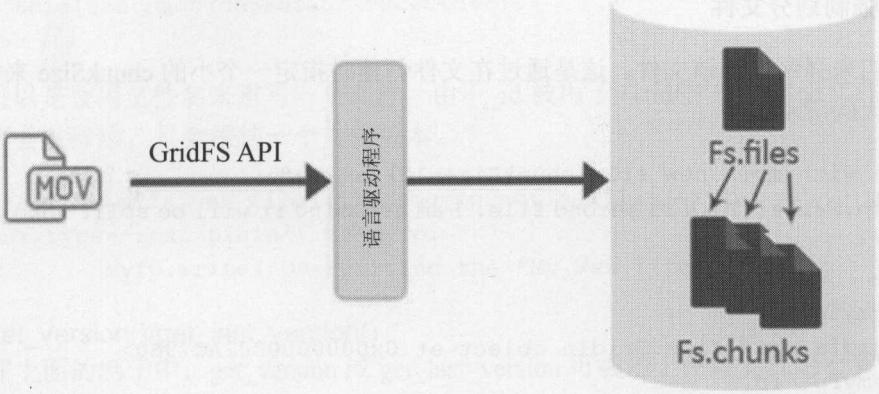

图 8-23 GridFS

8.6.3 使用 GridFS

在本节中，将使用 PyMongo 驱动程序来查看如何才能开始使用 GridFS。

1. 添加到文件系统的引用

需要做的第一件事就是添加到 GridFS 文件系统的引用：

```
>>> import pymongo
>>> import gridfs
>>>myconn=pymongo.Connection()
>>>mydb=myconn.gridfstest
>>>myfs=gridfs.GridFS(db)
```

write()

接下来你要执行一个基础写操作：

```
>>> with myfs.new_file() as myfp:
     myfp.write('This is my new sample file. It is just grand!')
```

find()

我们使用 mongo shell 来看看底层集合持有了什么：

```
    >>> list(mydb.myfs.files.find())
    [{u'length': 38, u'_id': ObjectId('52fdd6189cd2fd08288d5f5c'),
u'uploadDate': datetime.
    datetime(2014, 11, 04, 4, 20, 41, 800000), u'md5':
u'332de5ca08b73218a8777da69293576a',
    u'chunkSize': 262144}]
    >>> list(mydb.myfs.chunks.find())
    [{u'files_id': ObjectId('52fdd6189cd2fd08288d5f5c'), u'_id':
ObjectId('52fdd6189cd2fd08288d5 f5d'), u'data': Binary('This is my new
sample file. It is just grand!', 0), u'n': 0}]
```

2. 强制划分文件

我们来强制划分该文件。这是通过在文件创建时指定一个小的 chunkSize 来完成的，如以下代码所示：

```
>>> with myfs.new_file(chunkSize=10) as myfp:
    myfp.write('This is second file. I am assuming it will be split into various
chunks')
    >>>
    >>>myfp
    <gridfs.grid_file.GridIn object at 0x0000000002AC79B0>
    >>>myfp._id
    ObjectId('52fdd76e9cd2fd08288d5f5e')
    >>>list(mydb.myfs.chunks.find(dict(files_id=myfp._id)))
    ..................
    ObjectId('52fdd76e9cd2fd08288d5f65'), u'data': Binary('s', 0), u'n': 6}]
```

read()

现在你知道了文件实际上是如何被存储到数据库中的。接下来，使用客户端驱动程序，你将要读取该文件：

```
>>> with myfs.get(myfp._id) as myfp_read:
        printmyfp_read.read()
```

“这是另一个文件。我假设它将被划分成各个数据块。”

用户完全不必知道数据块。需要使用客户端公开的 API 从 GridFS 读取文件以及向其写入文件。

3. 要更像一个文件系统那样使用 GridFS

new_file()——在 GridFS 中创建一个新文件

可以将任意数量的关键字作为参数传递到 new_file()。它会被添加到 fs.files 文档中：

```
>>> with myfs.new_file(
        filename='practicalfile.txt',
        content_type='text/plain',
        my_other_attribute=42) as myfp:
                myfp.write('My New file')
    >>>myfp
    <gridfs.grid_file.GridIn object at 0x0000000002AC7AC8>
    >>>db.myfs.files.find_one(dict(_id=myfp._id))
    {u'contentType': u'text/plain', u'chunkSize': 262144,
u'my_other_attribute': 42,
    u'filename': u'practicalfile.txt', u'length': 8, u'uploadDate':
datetime.datetime(2014, 11, 04, 9, 01, 32, 800000), u'_id': ObjectId
('52fdd8db9cd2fd08288d5f66'), u'md5':
```

```
u'681e10aecbafd7dd385fa51798ca0fd6'}
>>>
```

可以是使用文件名来重写一个文件。由于_id 被用于 GridFS 中的索引文件，因此旧文件不会被移除。只会维护一个文件版本。

```
>>> with myfs.new_file(filename='practicalfile.txt',
content_type='text/plain') as myfp:
        myfp.write('Overwriting the "My New file"')
```

get_version()/get_last_version()

在上面的例子中，get_version 或 get_last_version 可被用于使用文件名检索文件。

```
>>>myfs.get_last_version('practicalfile.txt').read()
'Overwriting the "My New file"'
>>>myfs.get_version('practicalfile.txt',0).read()
'My New file'
```

也可以列出 GridFS 中的文件：

```
>>>myfs.list()
[u'practicalfile.txt', u'practicalfile2.txt']
```

delete()

也可以移除文件：

```
>>>myfp=myfs.get_last_version('practicalfile.txt')
>>>myfs.delete(myfp._id)
>>>myfs.list()
[u'practicalfile.txt', u'practicalfile2.txt']
>>>myfs.get_last_version('practicalfile.txt').read()
'My New file'
>>>
```

注意，这仅会移除 practicalfile.txt 的一个版本。在文件系统中你仍然具有一个名称为 practicalfile.txt 的文件。

exists()和 put()

接下来，要使用 exists()来检查一个文件是否存在以及使用 put()将一个短文件快速写入到 GridFS：

```
>>>myfs.exists(myfp._id)
False
>>>myfs.exists(filename='practicalfile.txt')
True
>>>myfs.exists({'filename':'practicalfile.txt'}) # equivalent to above
True
```

```
>>>myfs.put('The red fish', filename='typingtest.txt')
ObjectId('52fddbc69cd2fd08288d5f6a')
>>>myfs.get_last_version('typingtest.txt').read()
'The red fish'
>>>
```

8.7 索引

在本书的这一部分，将简要探究一个索引在 MongoDB 上下文中是什么样的。之后，我们将着重介绍 MongoDB 中可用的各种索引类型，本节会以重点介绍其行为和限制作为结束。

索引是一种可以加速读取操作的数据结构。通俗点来说，可以将其比作一本书的索引，其中可以通过查看章节的索引查找到任何章节，并且直接跳转到该页码，而不是扫描整本书来找到那一章，如果没有索引就需要这样做。

同样，索引也是在字段上定义的，这有助于以更好且更有效的方式搜索信息。

就像在其他数据库中一样，在 MongoDB 中也能看到一种类似的方式(它被用于加速 find()操作)。你运行的这类查询有助于为数据库创建有效的索引。例如，如果大多数查询都使用一个 Date 字段，那么在 Date 字段上创建一个索引将很有好处。弄明白哪个索引最适合你的查询是很难的，但这是值得一试的，因为如果准备好一个合适的索引，需要几分钟来返回结果的查询将即刻返回结果。

在 MongoDB 中，可以在文档的任意字段或子字段上创建一个索引。在你查看可以在 MongoDB 中创建的各种索引类型之前，我们先列出一些索引的核心特性：

- 索引是在每个集合这一级别上定义的。对于每一个集合来说，存在不同的索引集。
- 就像 SQL 索引一样，MongoDB 的索引也可以在单个字段或者一组字段上创建。
- 在 SQL 中，尽管索引增强了查询性能，但你要承担每一个写操作的开销。因此在创建任何索引之前，都应该考虑查询类型、频率、工作负荷大小，以及应用程序需求带来的插入量。
- 所有的 MongoDB 索引都使用了二叉树数据结构。
- 每一个使用更新操作的查询都仅会使用一个索引，这是由查询优化器决定的。这可以通过使用提示重写。
- 如果所有的字段都是索引的一部分，那么就可以说该索引覆盖了这个查询，无论该索引是用于查询还是用于投影。
- 覆盖索引最大化了 MongoDB 的性能和吞吐量，因为只要通过使用一个索引就能满足查询，而无须在内存中加载全部的文档。
- 只有在其上创建索引的字段发生变更时，索引才会被更新。并非文档上的所有更新操作都会造成索引变更。只有在关联字段受影响时它才会变更。

8.7.1 索引类型

在本节中，将了解 MongoDB 中可用的不同索引类型。

1. _id 索引

这是在_id 字段上创建的默认索引。这个索引无法被删除。

2. 辅助索引

用户在 MongoDB 中使用 ensureIndex()创建的所有索引都被称为辅助索引。

(1) 可以在文档或子文档中的任何字段上创建这些索引。我们思考以下文档：

```
{"_id": ObjectId(...), "name": "Practical User", "address":
{"zipcode": 201301, "state": "UP"}}
```

在这个文档中，索引可以被创建在 name 字段上，也可以被创建在 state 字段上。

(2) 这些索引可以被创建在持有一个子文档的字段上。

如果思考一下上面的文档，其中 address 持有一个子文档，在这种情况下，也可以在 address 字段上创建一个索引。

(3) 这些索引可以被创建在单个字段或者一组字段上。在使用一组字段创建索引时，它也被称为复合索引。

为了进一步解释它，我们思考一个持有以下格式文档的 products 集合：

```
{ "_id": ObjectId(...),"category": ["food", "grocery"], "item": "Apple",
"location": "16 th Floor Store", "arrival": Date(...)}
```

如果使用字段 Item 和 Location 的查询最多，则可以创建以下复合索引：

```
db.products.ensureIndex ({"item": 1, "location": 1})
```

除了涉及该复合索引所有字段的查询之外，上面的复合索引也可以支持使用任意索引前缀的查询(比如，它也可以支持仅使用 item 字段的查询)。

(4) 如果索引创建在一个持有数组及其值的字段上，那么一个多键索引就可用于单独索引该数组的每一个值。

思考以下文档：

```
{ "_id" : ObjectId("..."),"tags" : [ "food", "hot", "pizza", "may" ] }
```

标签上的索引就是一个多键索引，并且它将具有以下条目：

```
{ tags: "food" }
{ tags: "hot" }
{ tags: "pizza" }
{ tags: "may" }
```

(5) 也可以创建多键复合索引。不过，在任何时候，只有一个复合索引的字段可以是数组类型。

如果创建{a1: 1, b1: 1}这样的一个复合索引，则许可的文档就会如下面代码所示：

```
{a1: [1, 2], b1: 1}
{a1: 1, b1: [1, 2]}
```

以下文档是不允许的；实际上，MongoDB 甚至都无法插入这个文档：

```
{a1: [21, 22], b1: [11, 12]}
```

如果尝试插入这样一个文档，那么此插入行为将被拒绝并且会产生如下错误结果："无法索引平行数组"。

接下来将了解在创建索引时可能有用的各个选项/属性。

具有键排序的索引

MongoDB 索引会维护对字段的引用。这些引用要么是按照升序、要么是按照降序维护的。这是通过在创建一个索引时为键指定一个数字来完成的。这个数字表明了索引的方向。可能的选项是 1 和-1，其中 1 表示升序，而-1 表示降序。

在一个单键索引中，它可能并不十分重要；不过，在复合索引中，方向是非常重要的。

思考一个同时包含 username 和 timestamp 的 Events 集合。你的查询需要返回首先根据 username 在前，然后根据最近事件在前进行排序的事件。将会用到以下索引：

```
db.events.ensureIndex({ "username" : 1, "timestamp" : -1 })
```

这个索引包含了对按以下方式排序的文档的引用：

(1) 首先按照 username 字段升序排列。

(2) 然后按照 timestamp 字段降序排列每一个 username。

唯一索引

在你创建一个索引时，需要确保存储在索引字段中的值的唯一性。在这种情况下，可以通过将 Unique 属性设置为 true 来创建索引(默认情况下是 false)。

假设你希望在字段 userid 上有一个 unique_index，那么就可以运行以下命令来创建该唯一索引：

```
db.payroll.ensureIndex( { "userid": 1 }, { unique: true } )
```

这个命令会确保你在 user_id 字段上具有唯一值。在使用唯一性约束时，有几点需要注意：

- 如果在该场景下对一个复合索引使用唯一约束，则会在组合值上强制实现唯一性。
- 如果没有为唯一索引的字段指定值，则会存储一个空值。
- 在任何时候，只有一个文档被允许不使用唯一值。

dropDups

如果正在一个已经具有文档的集合上创建一个唯一索引，那么此创建行为可能会失败，因为可能已经存在一些文档在索引字段中包含重复值。在这种情况下，就可以使用 dropDups 选项来强制实现唯一索引的创建。这是通过保留第一个出现的键值而删除所有后续值来实现的。默认情况下 dropDups 是 false。

稀疏索引

稀疏索引是持有集合中文档条目的索引，这个集合的字段上已经创建了该索引。如果希望在 User 集合的 LastName 字段上创建一个稀疏索引，则可以运行以下命令：

```
db.User.ensureIndex( { "LastName": 1 }, { sparse: true } )
```

这个索引将包含如以下代码所示的文档：

```
{FirstName: Test, LastName: User}
```

或

```
{FirstName: Test2, LastName: }
```

不过，以下文档将不会成为该稀疏索引的一部分：

```
{FirstName: Test1}
```

该索引之所以被称为稀疏索引，是因为它只包含具有这些索引字段的文档，而不包含缺失这些字段的文档。由于其特性，稀疏索引带来了显著的空间节余。

相反，非稀疏索引包含了所有的文档，无论文档是否具有索引的字段。存储空值是为了防止字段缺失。

TTL 索引(生存时间，Time To Live)

在版本 2.2 中引入了一个新的索引属性，它允许你在经过特定时间段之后从集合中自动移除文档。这个属性适用于日志、会话信息以及机器生成的事件数据这样的场景，其中数据仅需要被持久化一段有限时间。

如果希望对集合 logs 设置一个小时的 TTL，则可以使用以下命令：

```
db.Logs.ensureIndex( { "Sample_Time": 1 }, { expireAfterSeconds: 3600} )
```

不过，需要注意以下限制：

- 在其上创建索引的字段必须仅仅是日期类型。在上面的例子中，字段 sample_time 必须持有日期值。
- 它不支持复合索引。
- 如果索引的字段包含具有多个日期的数组，那么文档会在该数组中最小日期匹配过期阈值时过期。
- 无法在已经创建了一个索引的字段上创建它。
- 这个索引无法被创建在固定集合上。

- TTL 索引会使用一个后台任务来让数据过期，该任务每分钟运行一次，以便移除过期的文档。因此你无法保证过期的文档不再存在于集合中。

地理位置索引

随着智能手机的兴起，查询当前位置附近的事情变得非常普遍。为了支持这样的基于位置的查询，MongoDB 提供了地理位置索引。

要创建一个地理位置索引，以下格式的坐标对必须存在于文档中：

- 要么是具有两个元素的数组
- 要么是具有两个键的嵌入式文档(键名称可以任意指定)。

下面是有效的示例：

```
{ "userloc" : [ 0, 90 ] }
{ "loc" : { "x" : 30, "y" : -30 } }
{ "loc" : { "latitude" : -30, "longitude" : 180 } }
{"loc" : {"a1" : 0, "b1" : 1}}.
```

可以使用以下命令在 userloc 字段上创建一个地理位置索引：

```
db.userplaces.ensureIndex( { userloc : "2d" } )
```

默认情况下，地理位置索引会假设值的范围是从-180 到 180。如果需要修改这个值，可以像下面这样使用 ensureIndex 来指定它：

```
db.userplaces.ensureIndex({"userloc" : "2d"}, {"min" : -1000, "max" :
1000})
```

任何具有超出最大和最小值的文档都会被拒绝。也可以创建复合地理位置索引。

我们用一个示例来理解这个索引是如何工作的。假设你有以下类型的文档：

```
{"loc":[0,100], "desc":"coffeeshop"}
{"loc":[0,1], "desc":"pizzashop"}
```

如果一个用户的查询是为了找出其位置附近所有的咖啡店，那么以下复合索引将会发挥作用：

```
db.ensureIndex({"userloc" : "2d", "desc" : 1})
```

地理堆索引

地理堆索引是基于存储桶的地理位置索引(也称为地理位置堆索引)。它们对于需要在小区域内找出各个位置同时又需要根据另一个维度进行过滤的查询来说很有用，比如找出坐标在 10 英里内并且类型字段值为 restaurant 的文档。

在定义索引时，需要强制指定 bucketSize 参数，因为它确定了对索引的粒度。例如，

```
db.userplaces.ensureIndex({ userpos : "geoHaystack", type : 1 },
{ bucketSize : 1 })
```

这个示例创建了一个索引，其中在 1 个单位纬度或经度以内的键会在相同存储桶中存储在一起。也可以在该索引中包含额外的类别，这意味着在查找位置详细信息的同时也会查找该信息。

如果用例通常是搜索“附近的”位置(比如，“25 英里内的餐厅”)，那么堆索引将更为有效。

可以在每个存储桶中找到并统计此额外索引字段(比如类别)的匹配。

相反，如果正在搜索“最近的餐厅”并且希望返回无论距离是多少的结果，那么一个普通的二维空间索引将更为有效。

目前(截至 MongoDB 2.2.0)堆索引有一些限制：

- 堆索引中只能包含一个额外字段。
- 这个额外的索引字段必须是单一值，而非数组。
- 不支持空的经度/纬度值。

除了上面提及的类型之外，版本 2.4 中还引入了一个新的索引类型，它支持对一个集合进行文本搜索。

之前在 2.6 的 beta 版本中，文本搜索是一个内置功能。它包括了像以 15 种语言进行搜索以及聚合选项这样的选项，它们可被用于设置根据产品或颜色的分面导航，例如用于一个电子商务网站上。

3. 索引交叉

索引交叉是在版本 2.6 中引入的，其中可以交叉多个索引来满足一个查询。为了进一步阐释它，我们思考一个持有以下格式文档的产品集合：

```
{ "_id": ObjectId(...),"category": ["food", "grocery"], "item": "Apple",
"location": "16 thFloor Store", "arrival": Date(...)}.
```

我们进一步假设这个集合具有以下两个索引：

```
{ "item": 1 }.
{ "location": 1 }.
```

上面两个索引的交叉可被用于以下查询：

```
db.products.find ({"item": "xyz", "location": "abc"})
```

可以运行 explain()来确定索引交叉是否被用于上述查询。解释输出结果将包含以下两个阶段之一：AND_SORTED 或 AND_HASH。在进行索引交叉时，可以使用整个索引或者仅使用索引前缀。

接下来需要理解这个索引交叉功能会如何影响复合索引的创建。

在创建一个复合索引时，该索引中排列键的顺序以及分类顺序(升序和降序)都很重要。因此复合索引可能不支持没有索引前缀或具有不同分类顺序的键的查询。

为了进一步阐释它，我们思考一个具有以下复合索引的产品集合：

```
db.products.ensureIndex ({"item": 1, "location": 1})
```

除了这个引用了复合索引所有字段的查询之外，上面的复合索引也可以支持正在使用任何索引前缀的查询(它也可以支持仅使用 item 字段的查询)。但它无法支持仅使用 location 字段或者使用具有不同分类顺序的 item 键的查询。

相反，如果创建两个独立的索引，一个位于 item 上，另一个位于 location 上，那么这两个索引就可以分别或者通过交叉来支持上面提到的 4 个查询。因此，是创建一个复合索引还是依赖索引交叉之间的选择取决于系统的需要。

注意，在 sort()操作需要一个与查询谓词完全无关的索引时，索引交叉将不适用。

例如，我们假设该产品集合有以下索引：

```
{ "item": 1 }.
{ "location": 1 }.
{ "location": 1, "arrival_date":-1 }.
{ "arrival_date": -1 }.
```

索引交叉将不会被用于以下查询：

```
db.products.find( { item: "xyz"} ).sort( { location: 1 } )
```

也就是说，MongoDB 不会将{ item: 1 }索引用于该查询，也不会将单独的{ location: 1 }或{ location: 1, arrival_date: -1 }索引用于分类排序。

不过，索引交叉可被用于以下查询，因为索引{location: 1,arrival_date: -1 }可以满足部分查询谓词：

```
db.products.find( { item: { "xyz"} , location: "A" } ).sort( { arrival_date:
-1 } )
```

8.7.2 行为和限制

最后，下面是需要知道的一些行为和限制：

- 集合中可能不允许使用多于 64 个索引。
- 索引键不能大于 1024 字节。
- 如果文档字段的值大于这个大小，那么这个文档就不能被索引。
- 以下命令可被用于查询太大而无法被索引的文档：

```
db.practicalCollection.find({<key>: <too large to index>})
.hint({$natural: 1})
```

- 索引名称(包括命名空间)必须小于 128 个字符。
- 插入/更新速度在一定程度上受到索引的影响。
- 不要维护不使用或将不再使用的索引。
- 由于$or 查询中的每一个子句都是平行执行的，因此每一个子句都可以使用一个不同的索引。

- 使用 sort()方法和$or 操作符的查询将无法使用$or 字段上的索引。
- 使用$or 操作符的查询不能得到别的地理位置查询的支持。

8.8 本章小结

在本章中，你了解了在底层数据是如何被存储的以及使用日志时写操作是如何进行的。你还了解了 GridFS 以及 MongoDB 中可用的不同索引类型。

在下一章中，将从管理视角来了解 MongoDB。

第 9 章

管理 MongoDB

“管理 MongoDB 不同于管理传统的 RDBMS 数据库。尽管大多数管理任务并非必要或者是由系统自动完成的，但仍然有一些任务需要人工介入。”

在本章中，将仔细了解用于备份和恢复、导入和导出数据、管理服务器，以及监控数据库实例的基本管理操作过程。

9.1 管理工具

在开始探究管理任务之前，这里有这些工具的一份快速概览。由于 MongoDB 没有提供 GUI 风格的管理界面，因此大多数管理任务都是使用命令行 mongo shell 来完成的。不过，一些独立的社区项目提供了一些可用的 UI。

9.1.1 mongo

mongo shell 是 MongoDB 发行版本的一部分。它是一个交互式的用于 MongoDB 数据库的 JavaScript shell。它为管理员提供了一个强大的接口，也为开发人员提供了测试查询以及直接操作数据库的接口。

在之前的章节中，你了解了使用 shell 进行的开发过程。在本章中，将使用 shell 来仔细了解系统管理任务。

9.1.2 第三方管理工具

有很多第三方工具可用于 MongoDB。大多数工具都是基于网络的。

10gen 在 MongoDB 网站 https://docs.mongodb.org/ecosystem/tools/administration-interfaces/ 上维护了支持 MongoDB 的所有第三方管理工具的清单。

9.2 备份和恢复

备份是其中一个最重要的管理任务。它会确保数据安全，以防出现任何紧急情况可

以恢复。

如果数据无法被恢复，那么备份就没有用了。因此，在进行一次备份之后，管理员需要确保它处于可用的格式，并且抓取的是一致状态的数据。

管理员需要学习的第一项技能就是如何进行备份以及将数据恢复。

9.2.1 数据文件备份

备份数据库最容易的方式是将数据复制到数据目录文件夹中。

MongoDB 所有的数据都会被存储到一个数据目录中，默认的目录是 C:\data\db(Windows 中)或者/data/db(Linux 中)。启动 mongod 时可以使用–dbpath 选项将这一默认路径修改成另一个目录。

数据目录内容是存储在 MongoDB 数据库中的数据全貌。因此使用 MongoDB 备份就是直接复制数据目录文件夹的整个内容。

通常，在 MongoDB 运行时复制数据目录内容并不安全。一种做法是在复制数据目录内容之前关闭 MongoDB 服务器。

如果服务器被正常关闭，那么数据目录的内容就代表了 MongoDB 数据的一份安全快照，因此可以在服务器再次重启之前复制它。

尽管这是进行备份的一种安全且有效的方式，但它并非理想的方式，因为它需要停机时间。

接下来，我们将探讨不需要停机时间进行备份的技术。

9.2.2 mongodump 和 mongorestore

mongodump 是 MongoDB 的备份工具，它是作为 MongoDB 发行版的一部分来提供的。通过查询一个 MongoDB 实例和将所有的读取文档写到硬盘，它就像一个普通客户端那样运行。

我们来执行一个备份，然后恢复它以验证该备份可用且格式一致。

以下代码片段来自于 Windows 平台上运行的工具。MongoDB 服务器运行在本地主机实例上。

打开一个终端窗口并且输入以下命令：

```
    C:\>cd c:\practicalmongodb\bin
    c:\practicalmongodb\bin>mongod --rest
    2015-07-15T22:26:47.288-0700 I CONTROL [initandlisten] MongoDB
starting :pid=3820
    port=27017 dbpath=c:\data\db\ 64-bit host=ANOC9
    ...................................................................
    2015-07-15T22:28:23.563-0700 I NETWORK [websvr] admin web console waiting
for connections on port 28017
```

为了运行 mongodump，请在一个新的终端窗口中执行以下命令：

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongodump
2015-07-15T22:29:41.538-0700 writing admin.system.indexes to
dump\admin\system.indexes.bson
.....
2015-07-14T22:29:46.720-0700 writing mydbproc.users to
dump\mydbproc\users.bson
c:\practicalmongodb\bin>
```

这会将整个数据库转储到 bin 文件夹目录本身的 dump 文件夹中，如图 9-1 所示。

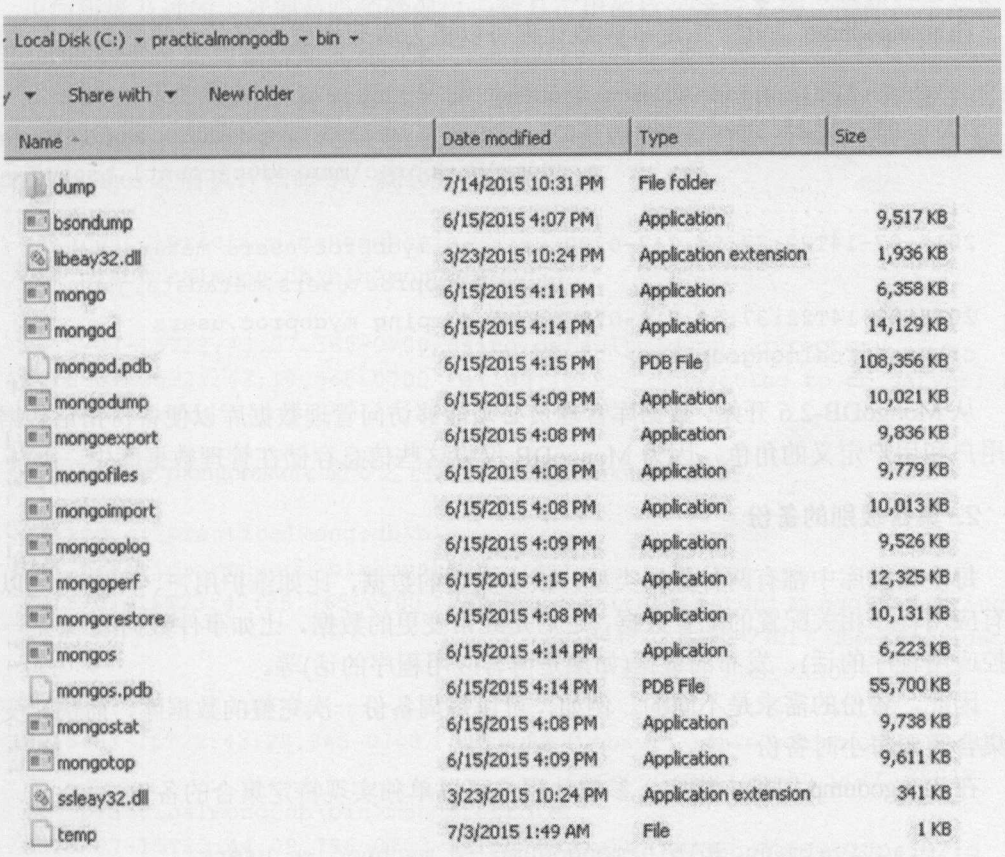

图 9-1 dump 文件夹

默认情况下，mongodump 工具会连接到默认端口上数据库的本地主机接口。

接下来，它会将每一个数据库以及与数据文件有关的集合拉取并且存储到一个预先定义好的文件夹结构中，默认为./dump/[databasename]/[collectionname].bson。

数据是以.bson 格式保存的，它类似于 MongoDB 用于在内部存储其数据的格式。

如果内容已经存在于该目录中，那么它将保持原封不动，直到转储包含相同的文件。例如，如果转储包含文件 c1.bson 和 c2.bson，并且输出目录包含文件 c3.bson 和 c1.bson，

那么 mongodump 就会用它的 c1.bson 文件替换该文件夹的 c1.bson 文件，并且将会复制 c2.bson 文件，但它不会移动或修改 c3.bson 文件。

在将其用于 mongodump 之前，你应该确保该目录为空，除非需要覆盖你的备份中的数据。

1. 单一数据库备份

在上面的示例中，你用默认设置执行了 mongodump，它会转储 MongoDB 数据库服务器上所有的数据库。

在真实的应用场景中，你的单台服务器上会运行多个应用程序数据库，每一个数据库都有不同的备份策略需求。

在 mongodump 工具中指定-d 参数让你可以更为明智地使用要备份的数据库。

```
c:\practicalmongodb\bin>mongodump -d mydbpoc
2015-07-14T22:37:49.088-0700 writing mydbproc.mapreducecount1 to
                             dump\mydbproc\mapreducecount1.bson
.....................
2015-07-14T22:37:54.217-0700 writing mydbproc.users metadata to
                             dump\mydbproc\users.metadata.json
2015-07-14T22:37:54.218-0700 done dumping mydbproc.users
c:\practicalmongodb\bin>
```

从 MongoDB-2.6 开始，数据库管理员必须能够访问管理数据库以便备份指定数据库的用户和用户定义的角色，因为 MongoDB 仅将这些信息存储在管理数据库中。

2. 集合级别的备份

每个数据库中都有两种数据类型：很少变更的数据，比如维护用户、用户角色以及所有应用程序相关配置的配置数据，之后是经常变更的数据，比如事件数据(如果是一个监控应用程序的话)、发布的数据(如果是博客应用程序的话)等。

因此，备份的需求是不同的。例如，可以每周备份一次完整的数据库，而快速变更的集合需要每小时备份一次。

在 mongodump 工具中指定-c 参数让用户可以单独实现特定集合的备份。

```
c:\practicalmongodb\bin>mongodump -d mydbpoc -c users
2015-07-14T22:41:19.850-0700 writing mydbproc.users to
                             dump\mydbproc\users.bson
2015-07-14T22:41:30.710-0700 writing mydbproc.users metadata to
                             dump\mydbproc\users.metadata.json
.....................................................
2015-07-14T22:41:30.712-0700 done dumping mydbproc.users
c:\practicalmongodb\bin>
```

如果没有指定数据需要被转储的文件夹，那么默认情况下，它会转储当前工作目录中名称为 dump 的目录中的数据，在这个例子中是 c:\practicalmongodb\bin。

3. mongodump –Help

你已经了解了执行 mongodump 的基础知识。除了上面提到的选项之外，mongodump 还提供了其他的选项，这些选项让你可以根据需求定制备份任务。就像使用所有其他工具一样，用–help 选项执行该工具将提供所有可用选项的列表。

4. mongorestore

正如所提及过的，管理员必须确保以一致且可用的格式进行备份。因此，下一步是使用 mongorestore 将数据转储恢复回来。

此工具会将数据库恢复到进行转储时的状态。在版本 3.0 之前，即使不启动 mongod/mongos，该命令也是可以运行的。不过从版本 3.0 开始，如果在启动 mongod/mongos 之前执行该命令，就会显示以下错误：

```
c:\>cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongorestore

2015-07-15T22:43:07.365-0700 using default 'dump' directory
2015-07-15T22:43:17.545-0700 Failed: error connecting to db server: no
reachable servers
```

必须在运行 mongorestore 命令之前运行 mongod/mongos 实例。

```
c:\>cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongod --rest
2015-07-15T22:43:25.765-0700 I CONTROL [initandlisten] MongoDBstarting :
                                    pid=3820 port=27017 dbpath=c:\data\
                                     db\ 64-bit host=ANOC9
.....................................................................
2015-07-15T22:43:25.865-0700 I NETWORK [websvr] admin web console waiting
                                     for connectionson port 28017
c:\ practicalmongodb\bin>mongorestore
2015-07-15T22:44:09.786-0700 using default 'dump' directory
2015-07-15T22:44:09.792-0700 building a list of dbs and collections to
restore from dump dir
.....................................
2015-07-15T22:44:09.732-0700 restoring indexes for collection
                             mydbproc.users from metadata
2015-07-15T22:44:09.864-0700 finished restoring mydbproc.users
c:\practicalmongodb\bin>
```

这样就会将数据强制附加到现有数据的后面。

要重写这一默认行为，应该在上述代码段中使用–drop。

–drop 命令向 mongorestore 工具表明，它需要删除前面提及的数据库中的所有集合和数据，然后将转储数据恢复到该数据库。

如果没有使用–drop，则该命令会将数据附加到现有数据的后面。

注意，从版本 3.0 开始，mongorestore 命令也可以接受来自标准输入的输入内容。

5. 恢复单个数据库

正如你在“备份”一节中所看到的，可以在单个数据库级别指定备份策略。可以通过使用-d 选项运行 mongodump 来进行单个数据库的备份。

同样，可以为 mongorestore 指定-d 选项来恢复单个数据库。

```
    c:\ practicalmongodb\bin>mongorestore
-d mydbpocc:\practicalmongodb\bin\dump\mydbproc -drop
    2015-07-14T22:47:01.155-0700 building a list of collections to restore
from
    C :\practicalmongodb\bin\dump\mydbprocdir
    2015-07-14T22:47:01.156-0700 reading metadata file from
    C :\practicalmongodb\bin\dump\mydbproc \users.metadata.json
    ..........................................................................
    2015-07-14T22:50:09.732-0700 restoring indexes for collection
                                 mydbproc.users from metadata
    2015-07-14T22:50:09.864-0700 finished restoring mydbproc.users
    c:\practicalmongodb\bin>
```

6. 恢复单个集合

正如可以在 mongodump 中使用-c 选项来指定集合级别的备份一样，你也可以将-c 选项与 mongorestore 工具一起使用，以恢复单个集合。

```
    c:\ practicalmongodb\bin>mongorestore -d mydbpoc -c users
    C:\ practicalmongodb\bin\dum\mydb\user.bson -drop

    2015-07-14T22:52:14.732-0700 restoring indexes for collection
                                 mydbproc.users from metadata
    2015-07-14T22:52:14.864-0700 finished restoring mydbproc.users
    c:\practicalmongodb\bin>
```

7. Mongorestore –Help

mongorestore 也有多个选项，可以使用-help 选项来查看它们。也可以参考这个网站：http://docs.mongodb.org/manual/core/backups/。

9.2.3 fsync 和锁

尽管上面这两个方法(mongodump 和 mongorestore)让你可以不需要停机就进行数据

库备份，它们没有提供获得时间点数据视图的能力。

你看到了如何复制数据文件以进行备份，但这需要在复制数据之前关闭服务器，而这在生产环境中是不可行的。

MongoDB 的 fsync 命令可以通过在不修改任何数据的情况下运行 MongoDB 来复制数据目录的内容。

fsync 命令会强制将所有等待的写操作刷新到硬盘。根据需要，它会持有一个锁，以阻止未来的写操作，直到服务器被解锁。就是这个锁才让 fsync 命令可用于备份。

要从 shell 中运行该命令，需要在一个新的终端窗口中连接到 mongo 控制台。

```
c:\practicalmongodb\bin>mongo
MongoDB shell version: 3.0.4
connecting to: test
>
```

接下来，切换到管理数据库并且使用 runCommand 运行 fsync：

```
>use admin
switched to db admin
>db.runCommand({"fsync":1, "lock":1})
{
        "info" : "now locked against writes, use db.fsyncUnlock() to unlock",
        "seeAlso" : " http://dochub.mongodb.org/core/fsynccommand ",
        "ok" : 1
}
>
```

此时，服务器已被锁定，不能进行任何写操作，从而确保数据目录代表着数据的一个一致的时间点快照。数据目录内容可以被安全地复制，以用作数据库备份。

必须在备份操作完成之后解锁数据库。为此，要运行以下命令：

```
>db.$cmd.sys.unlock.findOne()
{ "ok" : 1, "info" : "unlock completed" }
>
```

可以使用 currentOp 命令来检查数据库锁是否已经被释放。

```
>db.currentOp()
{ "inprog" : [ ] }
 (It may take a moment after the unlock is first requested.)
```

fsync 命令可以在不需要停机以及不需要牺牲备份时间点特性的情况下进行备份。不过，会存在短暂的写操作阻塞(也称为短暂的写操作停机)。

从版本 3.0 开始，在使用 WiredTiger 时，fsync 无法确保数据文件不会被修改。因此它无法被用于确保创建备份时的一致性。

接下来，将学习与从备份有关的内容。这是不需要任何停机就能使用时间点快照的

唯一一种备份技术。

9.2.4 从备份

从备份是 MongoDB 中数据备份的推荐方式。从节点总是会存储几乎与主节点保持同步的数据副本，并且从节点的可用性或性能并非一个太大的问题。可以将之前探讨过的任何技术应用到从节点，而非主节点上：关闭、带锁的 fsync，或者转储和恢复。

9.3 导入和导出

当尝试将你的应用程序从一个环境迁移到另一个环境中时，通常需要导入数据或导出数据。

9.3.1 mongoimport

MongoDB 提供了让你批量将数据直接加载到数据库的一个集合中的 mongoimport 工具。它从一个文件中读取并且将数据批量加载到一个集合中。

这些方法不适用于生产环境。

---

mongoimport 支持下面三种文件格式：

- **JSON**：在这种格式中，每行有一个 JSON 块，它表示一个文档。
- **CSV**：这是一种以逗号分隔的文件。
- **TSV**：TSV 文件与 CSV 文件相同；唯一的区别是，它使用制表符作为分隔符。

将–help 与 mongoimport 一起使用将提供该工具可用的所有选项。

mongoimport 非常简单。大多数时候你最终都会使用以下选项：

- **-h 或-host**：这会指定需要恢复数据的 mongod 的主机名称。如果未指定该选项，那么该命令默认将连接到在本地主机端口 27017 上运行的 mongod。根据需要，可以指定一个端口号来连接到运行在另一个端口上的 mongod。
- **-d 或-db**：指定需要导入数据的数据库。
- **-c 或-collection**：指定需要上传数据的集合。
- **--type**：这是文件的类型(比如 CSV、TSV 或 JSON)。
- **--file**：这是需要将数据导入到其中的文件路径。
- **--drop**：如果设置了这个选项，那么它将丢弃该集合并且从导入的数据中重建该集合。否则，数据会被附加到该集合的结尾处。
- **--headerLine**：这仅用于 CSV 或 TSV 文件，并且被用于表明第一行是头信息行。

以下命令会将数据从一个 CSV 文件导入到本地主机上的 testimport 集合：

```
c:\practicalmongodb\bin>mongoimport --host localhost -db mydbpoc
--collection testimport --typecsv -file c:\exporteg.csv --headerline
2015-07-14T22:54:08.407-0700   connected to:localhost
```

```
2015-07-14T22:54:08.483-0700  imported 15 documents
c:\ practicalmongodb\bin>
```

9.3.2 mongoexport

类似于 mongoimport 工具，MongoDB 提供了将数据从 MongoDB 数据库中导出的 mongoexport 工具。顾名思义，这个工具会从已有的 MongoDB 集合中导出文件。

使用–help 会显示 mongoexport 工具可用的选项。下面是你最终会用到的一些选项：

- **-q**：这个选项被用于指定将需要被导出的记录作为输出返回的查询。这类似于在你必须检索匹配选择条件的记录时在 db.CollectionName.find()函数中指定的内容。如果不指定查询，所有的文档都会被导出。
- **-f**：这个选项被用于指定需要从所选文档中导出的字段。

以下命令会将数据从 Users 集合导出到一个 CSV 文件：

```
c:\practicalmongodb\bin>mongoexport -d mydbpoc -c myusers -f _id,Age
                        -type=csv> myusers.csv
2015-07-14T22:54:48.604-0700  connected to: 127.0.0.1
2015-07-14T22:54:48.604-0700  exported 22 records
c:\practicalmongodb\bin>
```

9.4 管理服务器

在本节中，将了解作为系统管理员需要知道的各种选项。

9.4.1 启动一台服务器

本节将介绍如何启动服务器。之前，你使用了 mongo shell 通过运行 mongod.exe 来启动服务器。

可以通过在 Windows 中打开一个命令提示符(以管理员身份运行)或者在 Linux 系统上打开一个终端窗口并且输入以下命令来手动启动 MongoDB 服务器：

```
C:\>cd c:\practicalmongodb\bin
c:\ practicalmongodb\bin>mongod
mongod --help for help and startup options
.....................................
```

这个窗口将显示所有到该 mongod 的连接。它还会显示可被用于监控该服务器的信息。

如果没有指定任何配置，则 MongoDB 会使用 Windows 上的 C:\data\db 和 Linux 上的/data/db 这样的默认数据库路径来启动，并且使用默认端口 27017 和 27018 绑定到本地主机。

输入^C 将正常关闭该服务器。

MongoDB 提供了两个方法来指定用于启动服务器的配置参数。

第一个方法是使用命令行选项进行指定。

第二个方法是加载一个配置文件。可以通过编辑该文件然后重启服务器来修改服务器配置。

9.4.2 停止服务器运行

可以在其自身的 mongod 控制台中按下 CTRL+C 来关闭该服务器。或者，可以在 mongo 控制台中使用 shutdownServer 命令。

打开一个终端窗口，并且连接到 mongo 控制台。

```
C:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongo
MongoDB shell version: 3.0.4
connecting to: test
>
```

切换到管理 db 并且运行 shutdownServer 命令：

```
>use admin
switched to db admin
>db.shutdownServer()
2015-07-14T22:57:20.413-0700 I NETWORK DBClientCursor::init call() failed
server should be down...
2015-07-14T22:57:20.418-0700 I NETWORK trying reconnect to 127.0.0.1:27017
2015-07-14T22:57:21.413-0700 I NETWORK 127.0.0.1:27017 failed couldn't
connect to server 127.0.0.1:27017
>
```

如果检查上一步中启动服务器的 mongod 控制台，就会看到该服务器已经被成功关闭。

```
.....................
2015-07-14T22:57:30.259-0700 I COMMAND [conn1] terminating, shutdown
command received
2015-07-14T22:57:30.260-0700 I CONTROL [conn1] now exiting
.....................................................
2015-07-14T22:57:30.380-0700 I STORAGE [conn1] shutdown: removing fs
lock...
2015-07-14T22:57:30.380-0700 I CONTROL [conn1] dbexit: rc: 0
```

9.4.3 浏览日志文件

默认状态下，MongoDB 的完整日志输出会被写到 stdout，不过在启动服务器时通过在配置中指定 logpath 选项来修改它，以便将输出重定向到一个文件。

该日志文件的内容可被用于识别出像异常这样的问题，这些信息可以揭露一些数据问题或连接问题。

9.4.4 服务器状态

db.ServerStatus()是 MongoDB 提供的一个简单方法，它被用于检查服务器状态，比如连接数量、启动时间等。这个服务器状态命令的输出取决于操作系统平台、MongoDB 版本、所使用的存储引擎以及配置的类型(比如独立部署、副本集以及分片群集)。

从版本 3.0 开始，从输出结果中移除了后面这些内容：workingSet、indexCounters 以及 recordStats。

为了检查一台使用 MMAPv1 存储引擎的服务器的状态，需要连接到 mongo 控制台、切换到管理 db，并且运行 db.serverStatus()命令。

```
c:\practicalmongodb\bin>mongo
MongoDB shell version: 3.0.4
connecting to: test
>use admin
switched to db admin
>db.serverStatus()
host" : "ANOC9",
 "version" : "3.0.4",
 "process" : "mongod",
 "pid" : NumberLong(1748),
 "uptime" : 14,
 "uptimeMillis" : NumberLong(14395),
"uptimeEstimate" : 13,
 "localTime" : ISODate("2015-07-14T22:58:44.532Z"),
"asserts" : {
"regular" : 0,
        "warning" : 0,
        "msg" : 0,
        "user" : 1,
        "rollovers" : 0
},
.....................................................
```

上面的 serverStatus 输出也有“backgroundflushing”部分，它会显示对应于 MongoDB 用来将数据刷新到硬盘的程序的报告，其中 MongoDB 使用 MMAPv1 作为存储引擎。

“opcounters”和“asserts”部分提供了有用的信息，可用于分析以便界定任何可能的问题。

“opcounters”部分显示了每种类型操作的数量。为了找出是否存在任何问题，应该使用一条这些操作的基线。如果计数器开始偏离该基线，那么这就表示出现了问题并且

需要采取措施将它带回到正常状态。

“asserts”部分描述了客户端的数量以及发生的服务器警告或异常。如果发现出现了这样的异常和警告，那么就需要仔细查看日志文件以确认是否出现了问题。断言数量的增加可能也表示出现了数据问题，在这种情况下，应该使用 MongoDB 验证函数来检查数据是否正常。

接下来，使用 WiredTiger 存储引擎启动服务器并且查看 serverStatus 输出。

```
    c:\practicalmongodb\bin>mongod -storageEnginewiredTiger
    2015-07-14T22:51:05.965-0700 I CONTROL Hotfix KB2731284 or later update
is installed, no need to zero-out data files
    2015-07-29T22:51:05.965-0700 I STORAGE [initandlisten] wiredtiger_
openconfig:
    create,cache_size=1G,session_max=20000,eviction=(threads_max=4),
statistics=(fast),log=(enable ed=true,archive=true,path=journal,
compressor=snappy),file_manager=(close_idle_time=100000),
    checkpoint=(wait=60,log_size=2GB),statistics_log=(wait=0)
    ..................................................
```

为了检查服务器状态，需要连接到 mongo 控制台，切换到管理 db，并且运行 db.serverStatus()命令。

```
    c:\practicalmongodb\bin> mongo
    MongoDB shell version: 3.0.4
    connecting to: test
    > use admin
    switched to db admin
    >db.serverStatus()

    "wiredTiger" : {
    "uri" : "statistics:",
    "LSM" : {
    "..........................................................",
    "tree maintenance operations scheduled":0,
    ..........................................................,
    },
     "async" : {
              "number of allocation state races":0,
              "number of operation slots viewed for allocation":0,
              "current work queue length" : 0,
              "number of flush calls" : 0,
              "number of times operation allocation failed":0,
              "maximum work queue length" : 0,
    ..........................................................,
     },
     "block-manager" : {
```

```
            "mapped bytes read" : 0,
            "bytes read" : 966656,
            "bytes written" : 253952,
              ..................................,
            "blocks written" : 45
    },
    ..........................................................,
```

如你所见，在使用存储引擎 WiredTiger 启动时，该服务器状态输出结果有一个新的被称为 wiredTiger 统计信息的部分。

9.4.5 识别和修复 MongoDB

在本节中，将了解如何才能修复一个损坏的数据库。

如果得到了如下所示的错误：

- 数据库服务器拒绝启动，表明数据文件被损坏
- 断言出现在日志文件或者 db.serverStatus()命令中
- 奇怪或超出预期的查询结果

这意味着该数据库被损坏，并且必须运行修复以便恢复该数据库。

在可以启动修复之前，需要做的第一件事就是让服务器离线，如果它还未处于离线状态的话。可以使用上面提及的选项之一。在这个示例中，要在 mongod 控制台中输入 ^C。这样就会关闭该服务器。

接下来，使用–repair 选项启动 mongod，如下所示：

```
c:\practicalmongodb\bin>mongod --repair
2015-07-14T22:58:31.171-0700 I CONTROL Hotfix KB2731284 or later update
                                       is installed,no need to zero-out
                                       data files
2015-07-14T22:58:31.173-0700 I CONTROL [initandlisten] MongoDBstarting :
                                      pid=3996port=27017 dbpath=c:\data\
                                       db\ 64-bit host=ANOC9
2015-07-14T22:58:31.174-0700 I CONTROL [initandlisten] db version v3.0.4
..................................
2015-07-14T22:58:31.447-0700 I STORAGE [initandlisten] shutdown:
removing fs lock...
2015-07-14T22:58:31.449-0700 I CONTROL [initandlisten] dbexit: rc: 0
c:\ practicalmongodb\bin>
```

这样就会修复 mongod。如果查看该输出结果，就会发现该工具正在修复中的各种不一致。一旦修复过程完成，它就存在且可用了。

在修复过程完成之后，可以正常启动该服务器，然后可以将最近的数据库备份用于恢复丢失的数据。

有时候，你可能会注意到，当一个大的数据库处于修复状态时，该驱动的硬盘空间

正被耗尽。这是由于 MongoDB 需要在相同驱动上创建文件的临时副本作为数据文件这一原因造成的。要解决这个问题，在修复一个数据库时应该使用–repairpath 参数来指定修复过程期间可以用来创建临时文件的驱动。

9.4.6 识别和修复集合级别的数据

有时候你可能希望验证集合是否持有有效的数据以及有效的索引。对于这样的情况，MongoDB 提供了一个 validate()方法来验证指定集合的内容。

以下示例会验证 Users 集合的数据：

```
c:\practicalmongodb\bin>mongo
MongoDB shell version: 3.0.4
connecting to: test
>use mydbpoc
switched to dbmydbpoc
>db.myusers.validate()
{
        "ns" : "mydbpoc.myusers",
        "firstExtent" : "1:4322000 ns:mydbpoc.myusers",
        "lastExtent" : "1:4322000 ns:mydbpoc.myusers",
        "..............
        "valid" : true,
        "errors" : [ ],
        "warning" : "Some checks omitted for speed. use {full:true} option
    to domore thorough scan.",
        "ok" : 1
}
```

数据文件和相关的索引在默认情况下都会由 validate()选项进行检查。提供了集合统计信息以帮助识别数据文件或索引是否存在任何问题。

如果运行 validate()表明索引被损坏，那么在这种情况下就可以使用 reIndex 重新对该集合索引进行索引。这会丢弃并且重建该集合的所有索引。

以下命令会重新索引 Users 集合的索引：

```
>use mydbpoc
switched to dbmydbpoc
>db.myusers.reIndex()
{
        "nIndexesWas" : 1,
        "msg" : "indexes dropped for collection",
        "nIndexes" : 1,
        "indexes" : [
                {
                        "key" : {
```

```
                "_id" : 1
            },
            "ns" : "mydbpoc.myusers",
            "name" : "_id_"
        }
    ],
    "ok" : 1
}
>
```

如果该集合的数据文件被损坏，那么运行–repair 选项就是修复所有数据文件的最佳方式。

9.5 监控 MongoDB

作为一名 MongoDB 服务器管理员，重要的是监控该系统的性能和健康。在本节中，将了解监控该系统的几种方式。

9.5.1 mongostat

mongostat 是 MongoDB 发行版本的一部分。这个工具提供了关于服务器的简单统计信息；尽管它的内容并不丰富，但它提供了一份好的概要。后面的内容会显示本地主机的统计信息。打开一个终端窗口并且执行以下命令：

```
c:\> cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongostat
```

前 6 栏显示了 mongod 服务器以何种速率处理各种操作。除了这些栏之外，下面这一栏也值得注意，可以在诊断问题时使用它：

Conn： 这是连接到 mongod 实例的连接数量的指标。此处，一个较高的值可能表明了没有从应用程序释放或关闭连接的可能性，这意味着尽管应用程序发出了一个开启连接，但它并没有在操作完成之后关闭该连接。

从版本 3.0 开始，mongostat 也可以使用选项–json 来返回 json 格式的响应。

```
c:\>cd c:\practicalmongodb\bin
c:\practicalmongodb\bin>mongostat -json
```

```
{"ANOC9":{"ar|aw":"0|0","command":"1|0","conn":"1","delete":"*0","faul
ts":"1","flushes":"0", "getmore":"0","host":"ANOC9","insert":"*0","locked":"",
"mapped":"560.0M","netIn":"79b", "netOut":"10k","non mapped":"","qr|qw":
"0|0","query":"*0","res":"153.0M","time":"05:16:17","update":"*0","vsize":
"1.2G"}}
```

9.5.2 mongod 网络接口

无论何时启动了一个 mongod，它都会默认创建一个网络端口，该端口要比 mongod 用于侦听一个连接的端口号大 1000。默认情况下，该 HTTP 端口是 28017。

这个 mongod 网络接口是通过你的网络浏览器来访问的，并且它会显示大部分的统计信息。如果 mongod 运行在本地主机上并且正在侦听端口 27017 上的连接，那么可以使用 URL http://localhost:28017 来访问该 HTTP 状态页面。该页面看起来如图 9-2 所示。

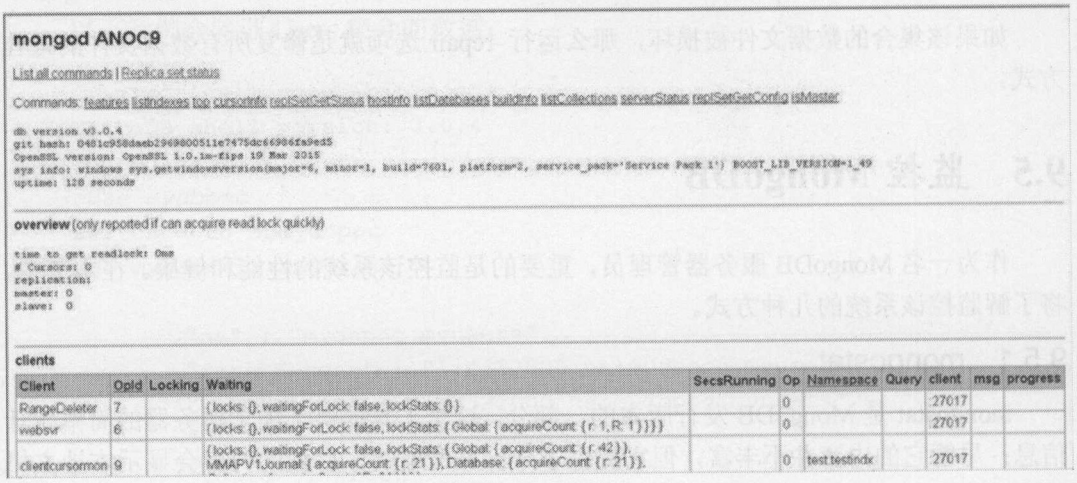

图 9-2 网络接口

9.5.3 第三方插件

除了这个工具之外，还有各种第三方适配器可用于 MongoDB，它们可以使用常见的开源或商业监控系统，比如 cacti、Ganglia 等。在其网站上，10gen 维护了一个页面，该页面分享了 MongoDB 可用的监控接口有关的最新信息。

要得到第三方插件的最新清单，可以查看 www.mongodb.org/display/DOCS/Monitoring+and+Diagnostics。

9.5.4 MongoDB 云管理器

除了上面谈到的用于监控和备份的工具和技术之外，还有 MongoDB 云管理器可用(其之前的名称为 MMS(MongoDB Monitoring Services，MongoDB 监控服务))。它是由开发 MongoDB 的团队开发出来的，并且可以免费使用(30 天试用许可)。与上面探讨的技术相反，MongoDB 云管理器提供了用户接口以及图表形式的日志和性能详情。

MongoDB 云管理器图表是交互式的，让用户可以设置一个自定义日期范围，如图 9-3 所描述的那样。

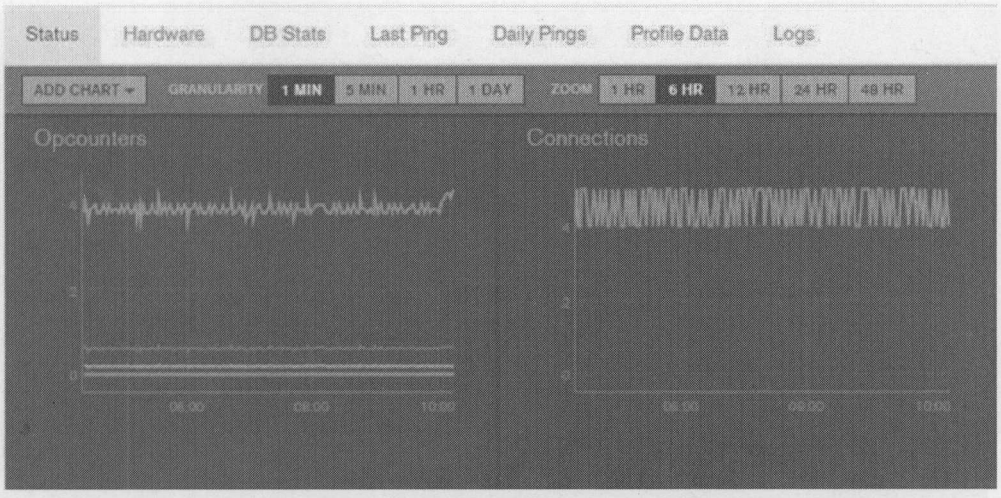

图 9-3 设置一个自定义日期范围

这个云管理器的另一个巧妙特性是出现不同事件时使用电子邮件以及文本告警的功能。图 9-4 中描述了这一点。

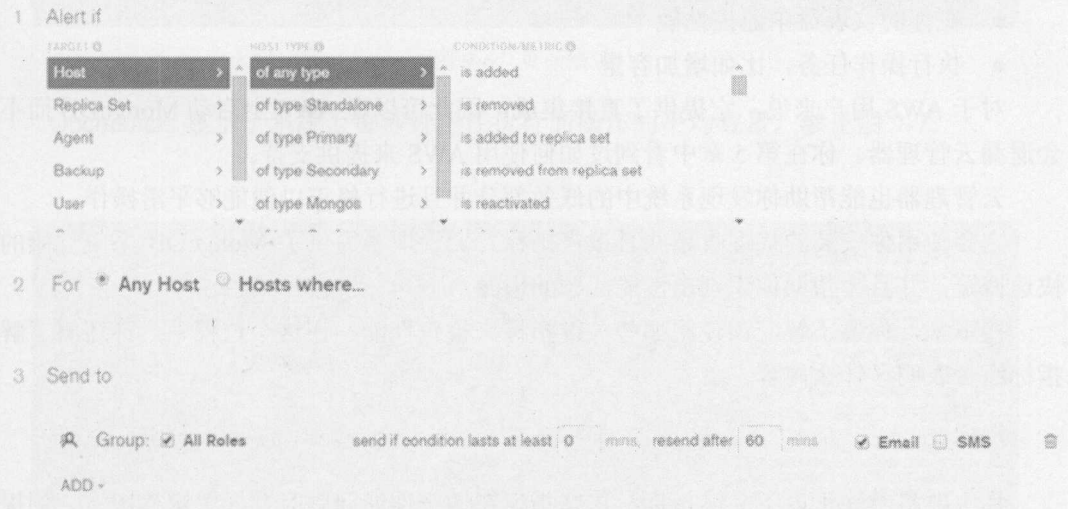

图 9-4 电子邮件和文本告警

云管理器不仅提供了图片和告警，它还让你可以浏览按照响应时间排序的较慢的查询。可以在一个地方轻易地看到你的查询执行得如何。图 9-5 显示了图表式查询性能的图片。

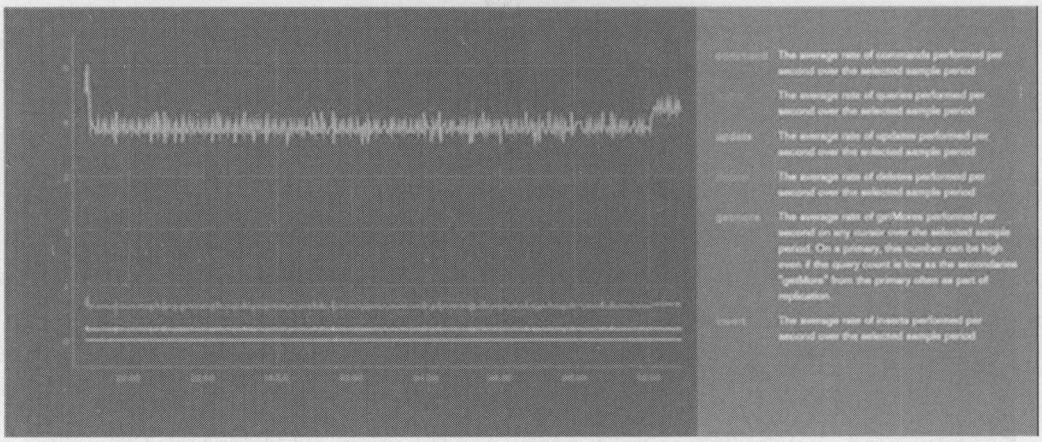

图 9-5 查询响应时间

云管理器让你可以执行以下操作：

- 自动化你的 MongoDB 部署(MongoDB 节点、群集以及现有部署升级的配置)
- 用连续的备份保护你的数据
- 提供使用 AWS 集成的任何拓扑结构
- 在你的仪表盘中监控性能
- 执行操作任务，比如增加容量

对于 AWS 用户来说，它提供了直接集成，因此可以在 AWS 上启动 MongoDB 而不会遗漏云管理器。你在第 5 章中看到过如何使用 AWS 来提供支持。

云管理器也能帮助你发现系统中的低效部分并且进行修正以便能够平滑操作。

它会使用你安装的代理收集并且报告指标。云管理器提供了 MongoDB 系统健康的快速概览，并且会帮助你识别出性能问题的根源。

接下来，你要了解应该使用哪些关键指标来检查性能。在这个过程中，你还将了解指标组合表明了什么内容。

指标

你主要将专注于以下关键指标；这些指标在调查性能问题时发挥关键作用。它们提供了关于 MongoDB 系统内部发生了什么以及哪个系统资源(比如 CPU、RAM 或硬盘)是瓶颈的快速概览。

- 页面错误
- Opcounters
- 锁的百分比
- 队列
- CPU 时间(IO 等待时间和用户时间)

要查看以下提及的图表，可以单击 Deployment 部分下面的 Deployment 链接。选择已经配置好的由云管理器监控的 MongoDB 实例。接着，在 Manage Charts 部分选择所需的图形/图表。

页面错误会显示系统中每秒发生的页面错误的平均数量。图 9-6 显示了页面错误图形。

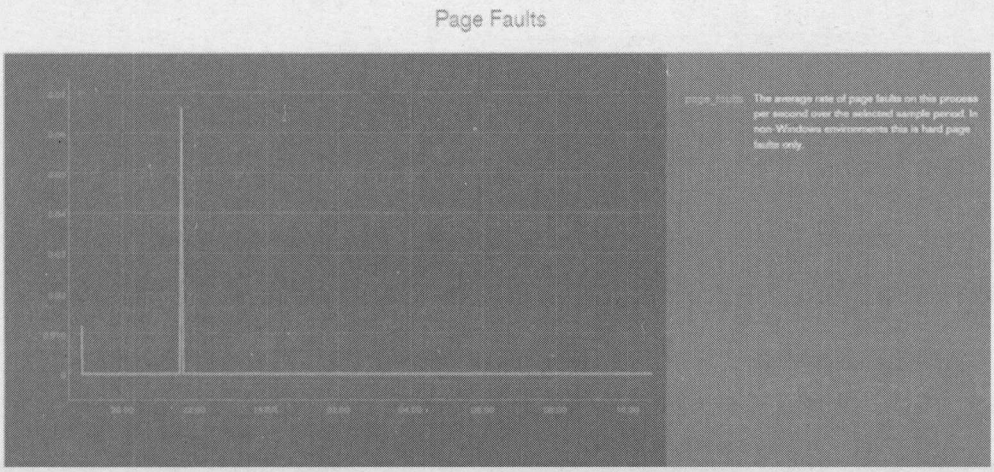

图 9-6 页面错误

Opcounters 显示了系统上每秒钟正在执行的操作的平均数量。参见图 9-7。

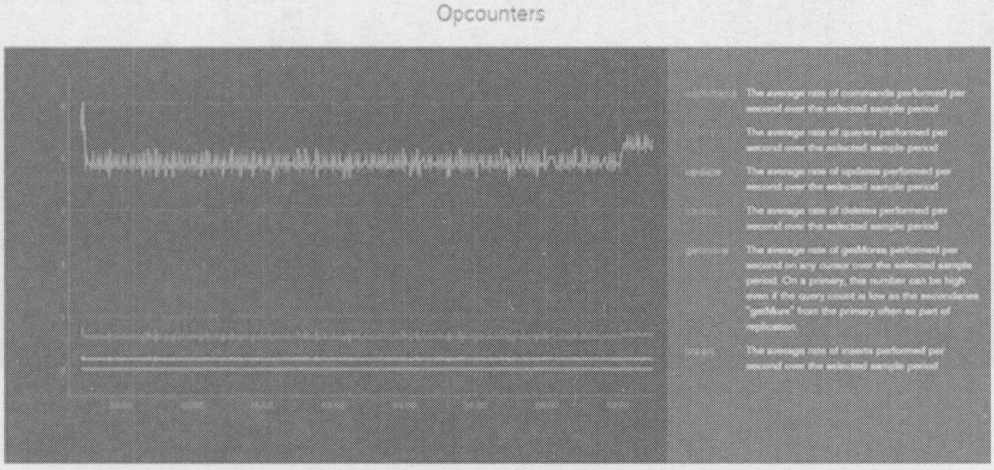

图 9-7 Opcounters

在页面错误与 Opcounters 的比率中，页面错误取决于正在系统上执行的操作以及当前位于内存中的内容。因此，每秒钟页面错误与每秒钟的 Opcounters 的比率可以提供硬盘 I/O 需求的真实情况。参见图 9-8。

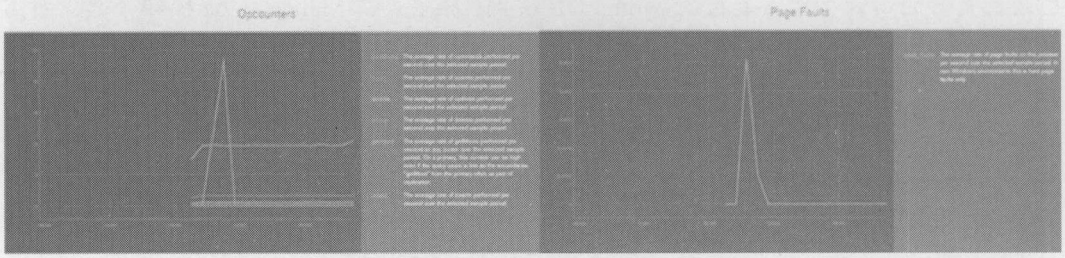

图 9-8 页面错误与 Opcounters 的比率

如果该比率是：

- <1，那么这会被归为低硬盘 I/O。
- 近似于 1，那么这会被归为正常的硬盘 I/O。
- >1，那么这会被归为高硬盘 I/O。

队列图形显示了在任意指定时点等待释放一个锁的操作计数。参见图 9-9。

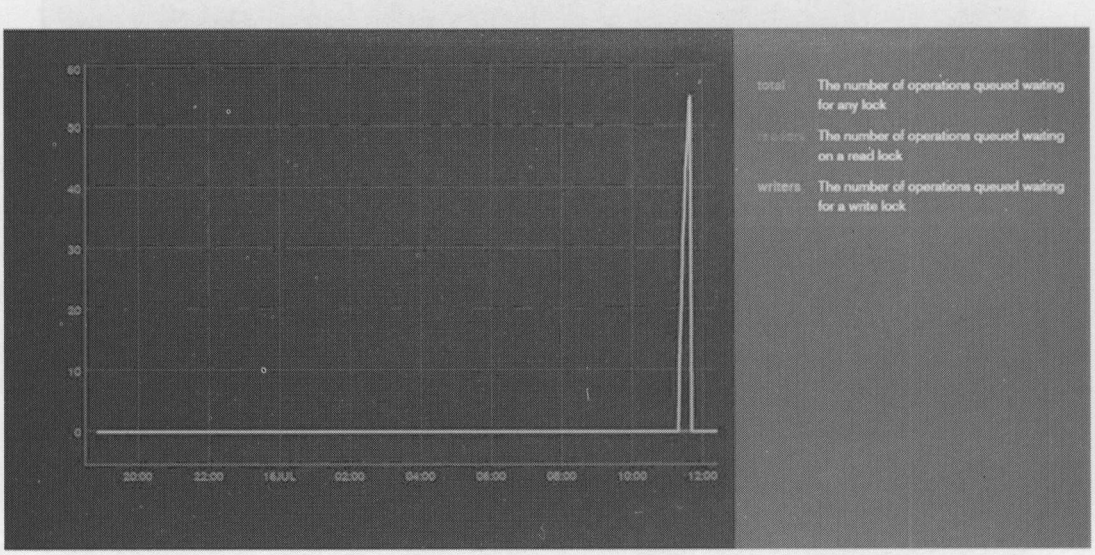

图 9-9 队列

CPU 时间(IO 等待时间和用户时间)图形显示了 CPU 核心是如何消耗其计算周期的。参见图 9-10。

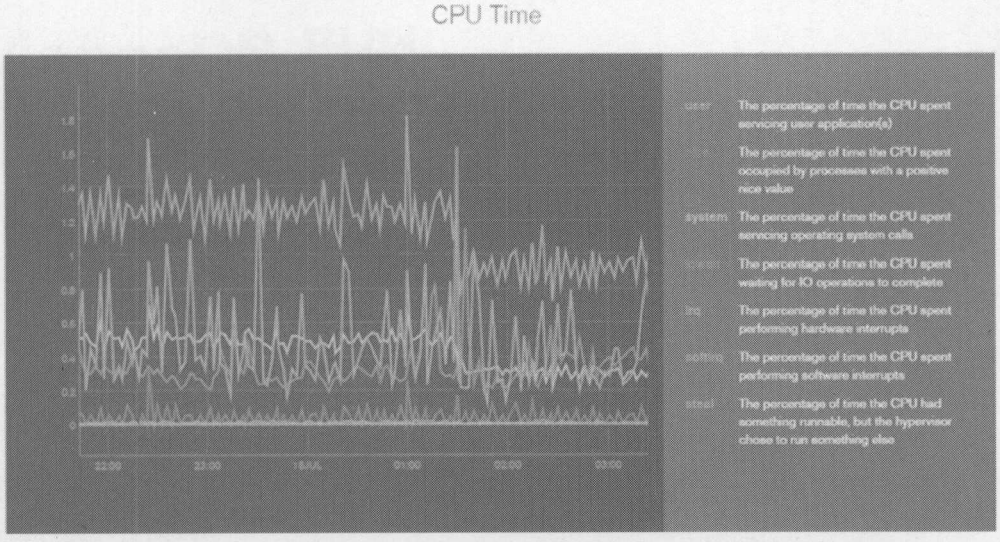

图 9-10　CPU 时间

IO 等待时间表明了 CPU 花在等待其他像硬盘或网络这样的资源上的时间。参见图 9-11。

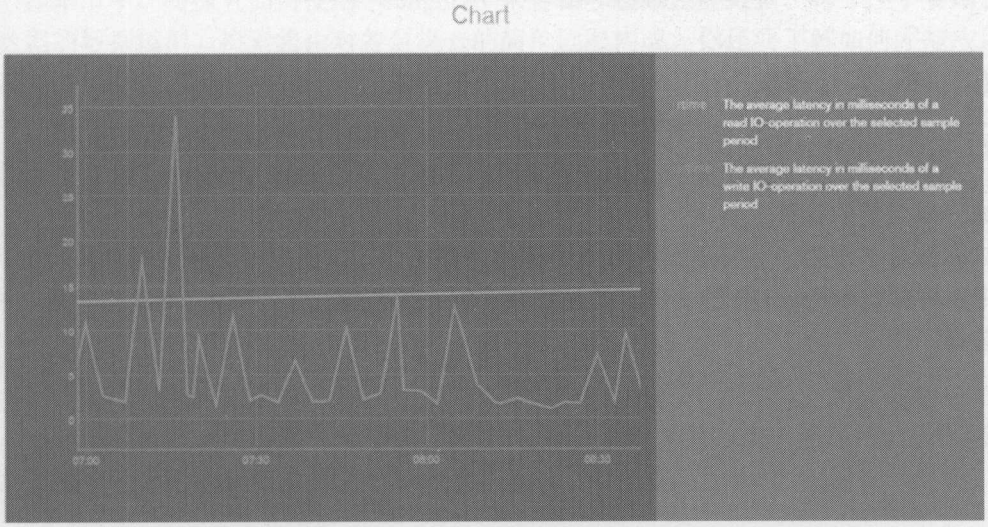

图 9-11　IO 等待时间

用户时间表明了花费在执行计算任务上的时间，比如文档更新、更新和重新平衡索引、选择或排序查询结果、运行聚合框架命令、运行 Map/Reduce，或者运行服务器端 JavaScripts。参见图 9-12。

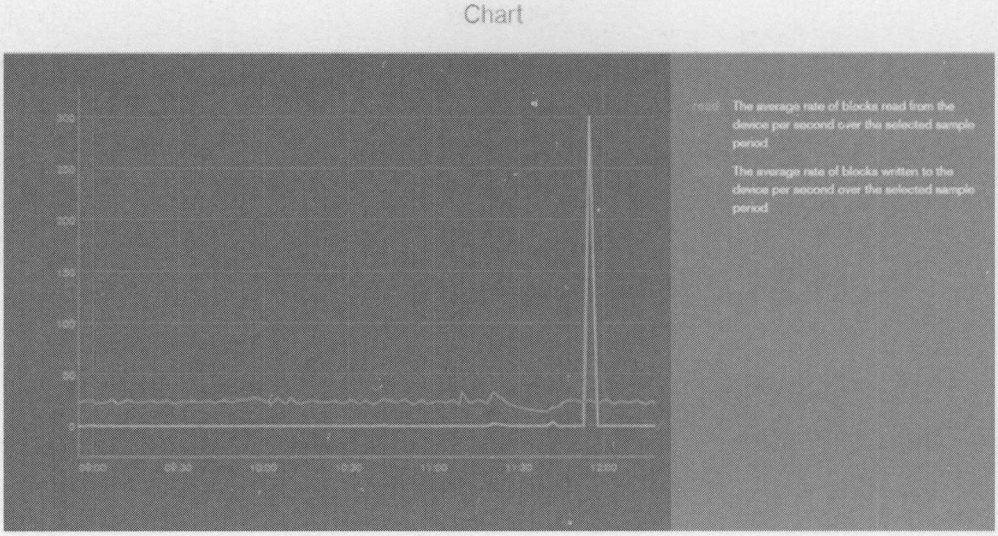

图 9-12 用户时间

要查看 CPU 时间图形，需要安装 munin。

应该使用这些关键指标及其组合来调查所有的性能问题。

9.6 本章小结

在本章中，你了解了如何使用被用于管理和维护系统的作为 MongoDB 发行版本的一部分来封装的各种工具。

你学习了作为一名管理员所必须了解的主要操作，以便深刻理解这些工具。在下一章中，将探究 MongoDB 的用例，并且还要了解在哪些情况下，MongoDB 并非是一种好的选择。

第 10 章

MongoDB 用例

“MongoDB：它对于我来说有没有用？”

在本章中，我们会把 MongoDB 的功能与它适合解决的业务问题之间联系起来，以便进行你肯定已经渴望已久的这方面的探讨。我们将使用两个用例来分享用于处理此类问题的技术和模式。

10.1 用例 1——性能监控

在本节中，将探究如何使用 MongoDB 来存储和检索性能数据。将专注于要用于存储数据的数据模型。检索将由对各自集合的简单读取构成。你还要了解如何才能应用分片和复制以便获得更好的性能以及数据安全性。

我们假设有一个监控工具正在收集 CSV 格式的服务器定义的参数数据。通常，监控工具要么会在服务器上的 designated 文件夹中将数据存储为文本文件，要么它们会将其输出重定向到任意报告数据库服务器。在这个用例中，会有一个调度器读取这一共享文件夹路径并且将数据导入到 MongoDB 数据库中。

10.1.1 模式设计

设计一个解决方案的第一步是决定其模式。模式取决于监控工具要捕获的数据格式。日志文件中的一行可能类似于表 10-1。

表 10-1 日志文件

节点 UUID	IP 地址	节点名称	MIB	时间戳(毫秒)	测量值
3beb1a8b-040d-4b46-932a-2d31bd353186	10.161.1.73	corp_xyz_sardar	IFU	1369221223384	0.2

如下代码所示是将每一行存储为文本的最简单方式：

```
{
```

```
    _id: ObjectId(...),
    line: '10.161.1.73 - corp_xyz_sardar [15/July/2015:13:55:36 -0700]
"Interface Util" ...
    }
```

尽管这会捕获到数据，但它对于用户来说没有任何意义，因此，如果希望找出来自特定服务器的事件，就需要使用正则表达式，它会引发对集合的完全扫描，而这是非常低效的。

相反，可以从日志文件中提取数据并且将它存储为 MongoDB 文档中有意义的字段。

注意，在设计该结构时，使用正确的数据类型非常重要。这不仅会节省空间，还会对性能产生重大影响。

例如，如果将日志的日期和时间字段存储为一个字符串，那么它就不仅会使用更多的字节，还会难以发起日期范围的查询。相较于使用字符串，如果将日期存储为一个 UTC 时间戳，那么它就需要 8 个字节，而非用于字符串的 28 个字节，因此执行日期范围查询会比较容易。如你所见，将合适的类型用于数据会提高查询的灵活性。

需要将以下文档用于存储你的监控数据：

```
{
id: ObjectID(...),
Host:,
Time:ISODate(''),
ParameterName:'aa',
Value:10.23
}
```

提示： 实际的日志数据可能具有额外的字段；如果完整捕获它，则可能会得到一个较大的文档，这将是对存储和内存的低效使用。在设计模式时，你应该忽略不需要的详细信息。识别出你必须捕获哪些字段以满足你的需求是非常重要的。

在你的场景中，满足你的报告应用程序需求的最重要信息如下所示：

(1) 主机
(2) 时间戳
(3) 参数
(4) 值

10.1.2 操作

在设计好文档结构之后，接下来你要了解需要在系统上执行的各种操作。

1. 插入数据

用于插入数据的方法取决于你的应用程序的写关注。

(1) 如果正在寻求快速的插入速度并且允许在数据安全性方面做出让步，则可以使

用以下命令：

```
>db.perfpoc.insert({Host:"Host1", GeneratedOn: new
ISODate("2015-07-15T12:02Z"),ParameterName:"CPU",Value:13.13},w=0)
>
```

尽管这个命令是可用的最快选项，但由于它不会等待操作是否成功的任何确认，所以会有丢失数据的风险。

(2) 如果只是想确认至少数据被保存好了，那么可以运行以下命令：

```
>db.perfpoc.insert({Host:"Host1", GeneratedOn: new
ISODate("2015-07-15T12:07Z"),ParameterName:"CPU",Value:13.23},w=1)
>
```

尽管这个命令会确认数据被保存了，但它不会提供任何针对数据丢失的安全保障，因为它并非日志式的。

(3) 如果主要关注的是牺牲插入速度的提升而得到数据安全性保障，那么可以运行以下命令：

```
>db.perfpoc.insert({Host:"Host1", GeneratedOn: new
ISODate("2015-07-15T12:09Z"),ParameterName:"CPU",Value:30.01},j=true,w=2)
>
```

在这段代码中，不仅要确保数据被复制，还要启用日志。除了复制确认之外，它还会等待一个成功的日志提交。

提示：尽管这是最安全的选项，但它也会对插入性能产生严重影响，因此它是速度最慢的操作。

---

2. 批量插入

在使用严格的写关注时，批量插入事件总是有益的，就你的情况而言，这是因为它使得 MongoDB 可以将所引起的性能损失分散到一组插入中。

如果可能，应该将批量插入用于插入监控数据，因为这些数据会很大并且会在数秒钟内生成。将它们作为一个组来分组到一起并且进行插入将带来较好的影响，因为在相同的等待时间内，会有多个事件被保存。因此对于这个用例来说，你要使用批量插入来分组多个事件。

3. 查询性能数据

你已经看到了如何插入事件数据。当能够通过查询数据来响应特定查询时，维护数据的价值就凸显出来了。

例如，你可能希望查看与特定字段相关的所有性能数据，比如 Host。

将了解用于抓取数据的几种查询模式，然后你要了解如何优化这些操作。

查询 1：抓取特定主机的性能数据

```
>db.perfpoc.find({Host:"Host1"})
{ "_id" : ObjectId("553dc64009cb76075f6711f3"), "Host" : "Host1",
"GeneratedOn":
ISODate("2015-07-18T12:02:00Z"), "ParameterName" : "CPU", "Value" :
13.13 }
{ "_id" : ObjectId("553dc6cb4fd5989a8aa91b2d"), "Host" : "Host1",
"GeneratedOn":
ISODate("2015-07-18T12:07:00Z"), "ParameterName" : "CPU", "Value" :
13.23 }
{ "_id" : ObjectId("553dc7504fd5989a8aa91b2e"), "Host" : "Host1",
"GeneratedOn":
ISODate("2015-07-18T12:09:00Z"), "ParameterName" : "CPU", "Value" :
30.01 }
>
```

如果需求是要分析一个主机的性能，则可以使用它。

在 Host 字段上创建一个索引将优化上述查询的性能：

```
>db.perfpoc.ensureIndex({Host:1})
>
```

查询 2：抓取日期范围为 2015 年 7 月 10 日～2015 年 7 月 20 日的数据

```
>db.perfpoc.find({GeneratedOn:{"$gte": ISODate("2015-07-10"), "$lte":
ISODate("2015-07-20")}})
{ "_id" : ObjectId("5302fec509904770f56bd7ab"), "Host" : "Host1",
"GeneratedOn"
.....
>
```

如果希望思考并且分析收集到的特定日期范围内的数据，这就很重要。在这个例子中，“时间”上的索引将对性能产生积极影响。

```
>db.perfpoc.ensureIndex({GeneratedOn:1})
>
```

查询 3：抓取日期范围为 2015 年 7 月 10 日～2015 年 7 月 20 日的特定主机的数据

```
>db.perfpoc.find({GeneratedOn:{"$gte": ISODate("2015-07-10"), "$lte":
ISODate("2015-07-20")}, Host: "Host1"})
{ "_id" : ObjectId("5302fec509904770f56bd7ab"), "Host" : "Host1",
"GeneratedOn"
.....
>
```

如果希望查看特定时间段内一个主机的性能数据，这就很有用。

在像这样的涉及多个字段的查询中，所使用的索引会对性能产生显著影响。例如，对于上述查询来说，创建一个复合索引将很有好处。

还需要注意的是，复合索引中字段的顺序也会产生影响。我们用一个示例来理解其差异。我们来创建一个如下所示的复合索引：

```
>db.perfpoc.ensureIndex({"GeneratedOn":1,"Host":1})
>
```

接下来，对其进行阐释：

```
>db.perfpoc.find({GeneratedOn:{"$gte": ISODate("2015-07-10"), "$lte":
ISODate("2015-07-20")}, Host: "Host1"}).explain("allPlansExecution")
.....................................................................
"allPlansExecution" : [
        {
                "nReturned" : 4,
                "executionTimeMillisEstimate" : 0,
                "totalKeysExamined" : 4,
                "totalDocsExamined" : 4
                "indexName" : "GeneratedOn_1_Ho
.....................................................................
                              "isMultiKey" : false,
                              "direction" : "forward",
      }]
.....................................................................
```

丢弃该复合索引，如下所示：

```
>db.perfpoc.dropIndexes()
{
        "nIndexesWas" : 2,
        "msg" : "non-_id indexes dropped for collection",
        "ok" : 1
}
```

或者，反向使用字段创建复合索引：

```
>db.perfpoc.ensureIndex({"Host":1,"GeneratedOn":1})
>
```

进行阐释：

```
>db.perfpoc.find({GeneratedOn:{"$gte": ISODate("2015-07-10"), "$lte":
ISODate("2015-07-20")}, Host: "Host1"}).explain("allPlansExecution")
{
        ........................................
        "executionStats" : {
        "executionSuccess" : true,
```

```
        "nReturned" : 4,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 4,
        "totalDocsExamined" : 4,
.....................................................
"allPlansExecution" : [ ]
.....................................................
}
>
```

可以看到该阐释命令的输出结果中的差异。

使用 explain()，可以弄明白索引的影响并且基于你的应用程序用途相应地就索引做出决定。

还建议使用单个复合索引来覆盖最多的查询，而不是使用多个单键索引。

基于你的应用程序用途以及该阐释统计信息的结果，你只要在{'GeneratedOn':1, 'Host': 1}上使用一个复合索引来覆盖上面提到的所有查询即可。

查询 4：抓取按照主机和日期计数的性能数据

列出数据很好，但对于性能数据，最常用于执行的查询是得到计数、平均数、合计数，或者分析期间的其他聚合操作。此处将看到如何使用 aggregate 命令对结果进行选择、处理和聚合以便满足强大的即席查询的需要。

为了进一步阐释说明，我们来编写一个计算每月数据量的查询：

```
>db.perfpoc.aggregate(
... [
... {$project: {month_joined: {$month: "$GeneratedOn"}}},
... {$group: {_id: {month_joined: "$month_joined"}, number: {$sum:1}}},
... {$sort: {"_id.month_joined":1}}
... ]
... )
{ "_id" : { "month_joined" : 7 }, "number" : 4 }
>
```

为了优化其性能，需要确保筛选字段具有一个索引。你已经创建了一个覆盖相同字段的索引，因此目前不需要为这一场景创建任何额外的索引。

10.1.3 分片

性能监控数据集是非常巨大的，因此它迟早会超出单台服务器的容量。这样一来，你就应该考虑使用一个分片群集。

在本节中，将了解哪个分片键真正适合性能数据的用例，这样其负荷才会跨群集分散并且没有一台服务器会超负荷。

分片键控制数据分布方式以及影响用于查询和写入的系统容量。理想情况下，分片

键应该具有以下两种特征：

- 插入是跨分片群集平衡的。
- 大多数查询都可以被路由到需要满足的分片的一个子集。

下面介绍可以将哪些字段用于分片。

(1) **时间字段**：在选择这个选项时，尽管数据将会跨分片平均分布，但无论是插入还是读取都不会被平衡。

例如在性能数据的例子中，时间字段处于向上的方向，因此所有的插入最终都会去到单个分片，而写入的吞吐量最终也同样会在一个独立的实例中。

如果希望频繁浏览最近的数据，那么大多数读取也将最终在相同分片上进行。

(2) **哈希值**：你也可以考虑使用一个随机值来满足上述情形的需要；可以考虑将_id 字段的一个哈希值作为分片键。

尽管这将会满足上面写入情况的需要(也就是说，写入操作将被分布)，但它会影响到查询。在这种情况下，查询必然被散播到所有的分片，并且将不可被路由。

(3) 使用键，它是平均分布的，比如**主机**。

这样就能带来以下好处：如果查询选择主机字段，那么对于单个分片来说，读取操作将是选择性且局部的，而写入操作将被平衡。

不过，最大的可能缺陷在于，收集到的单个主机的所有数据都必须位于相同数据块，因为其中所有的文档都具有相同的分片键。如果是跨所有主机收集的数据，那么这将不是问题，但如果监控为一个主机收集了过多的数据，那么你最终将得到一个很大的数据块，它将完全无法被划分，从而造成分片的负荷失衡。

(4) 合并选项 2 和 3 的最佳部分，可以使用一个复合分片键，比如{host:1, ssk: 1}，其中 host 是文档的主机字段，而 ssk 是_id 字段的哈希值。

在这种情况下，数据很大程度上是根据进行查询的主机字段来分布的，这些查询会在局部的一个分片或者一组分片上访问该主机字段。同时，使用 ssk 会确保数据跨群集均匀分布。

在大多数情况下，这样的键不仅会确保写操作跨群集理想分布，还会确保查询仅访问为其指定的若干分片。

尽管如此，最佳的方式是分析应用程序的真实查询和插入，然后选择一个合适的分片键。

10.1.4 管理数据

由于性能数据很庞大，并且它会持续增长，因此可以定义一个数据保留策略，它会表明你要将数据保留的一个特定时间段(比如 6 个月)。

那么你要如何移除老的数据呢？可以使用以下方式：

- **使用一个固定集合**：尽管固定集合可被用于存储性能数据，但对固定集合进行分片是不可能的。

- **使用一个 TTL 集合：**此方式会创建一个类似于固定集合的集合，但它可以被分片。
  在这种情况下，可以在该集合上定义一个生存时间索引，它会让 MongoDB 可以定期从该集合中移除旧文档。不过，这样做并不会具有固定集合的性能优势；此外，remove()操作还可能会导致数据碎片。
- **使用多个集合存储数据：**第三种方式是每天创建一个集合，该集合会包含存储当天性能数据的文档。这样一来，你最终在一个数据库中就会具有多个集合。尽管这样做会让查询复杂化(为了抓取两天的数据，你可能需要读取两个集合)，但删除一个集合会很快，并且空间可以被有效重复使用，而不会产生任何数据碎片。
  在你的用例中，你要将这一模式用于管理数据。

10.2 用例 2——社交网络

在本节中，将探究如何使用 MongoDB 来存储和检索一个社交网络站点的数据。

这个用例大致上是一个友好的社交网络站点，它允许用户分享其状态和照片。为这个用例提供的解决方案假设了以下情形：

(1) 用户可以选择是否关注另一个用户。

(2) 用户可以决定他希望与之分享更新的用户圈。
这个圈的选项有好友、好友的好友以及公开。

(3) 所允许的更新有状态更新和照片。

(4) 用户个人信息会显示兴趣、性别、年龄以及关系状态。

10.2.1 模式设计

你要提供的解决方案，其目标是最小化为了显示任何指定页面而必须加载的文档的数量。讨论中的应用程序具有两个主要页面：第一个页面会显示用户墙(它旨在呈现某个用户创建的发布内容或者指向该用户的发布内容)，另一个页面是社交新闻页面，它会显示关注该用户或者该用户关注的所有人的全部通知和活动内容。

除了这两个页面之外，还有一个用户个人信息页面，它会显示与该用户个人信息有关的详细信息，以及他的好友组的信息(那些关注他或他关注的人)。为了满足这一需求，用于这个用例的模式要由以下集合构成。

第一个集合是 user.profile，它会存储与用户个人信息有关的数据：

```
{
_id: "User Defined unique identifier",
UserName: "user name"
ProfilDetaile:
            {Age:.., Place:.., Interests: ...etc},
```

```
FollowerDetails: {
                  "User_ID":{name:..., circles: [circle1, circle2]}, ....
                },
CirclesUserList: {
                "Circle1":
                      {"User_Id":{name: "username"}, ......
                      }, .......
                }       ,
      ListBlockedUserIDs: ["user1",...]
}
```

- 在这个例子中，你手动指定了_id 字段。
- Follower 列出了关注该用户的用户。
- CirclesUserList 由此用户关注的圈子构成。
- Blocked 由被此用户阻止浏览其更新的用户构成。

第二个集合是 user.posts 集合，使用以下模式：

```
{
_id: ObjectId(...),
by: {id: "user id", name: "user name"},
VisibleTocircles: [],
PostType: "post type",
ts: ISODate(),
Postdetail: {text: "",
Comments_Doc:
 [
{Commentedby: {id: "user_id", name: "user name"}, ts: ISODate(),
Commenttext: "comment text"}, .....
]
}
```

- 这个集合用于显示该用户的所有活动。by 提供了发布该帖子的用户的信息。Circles 控制了该帖子对其他用户是否可见。Type 用于识别帖子的内容。ts 是帖子的创建时间。detail 包含了帖子的文本，并且其中具有嵌入的评论。
- comment 文档由以下详细信息构成：by 提供了对该帖子进行评论的用户 id 和姓名的详细信息，ts 是评论的时间，而 text 是该用户发布的实际评论。

第三个集合是 user.wall，它用于呈现第二个集合的一个片段中的用户的墙页面。这个集合会从第二个集合抓取数据并且以概要的方式存储该数据以便能够快速呈现墙页面。

该集合具有以下格式：

```
{
_id: ObjectId(...),
User_id: "user id"
```

```
    PostMonth: "201507",
    PostDetails: [
    {
    _id: ObjectId(..), ts: ISODate(), by: {_id: .., Name:.. }, circles: [..],
type: ....
    , detail: {text: "..."}, comments_shown: 3
    ,comments: [
    {by: {_id:., Name:....}, ts: ISODate(), text:""}, ......]
    },....]}
```

- 如你所见，你正在根据每用户每月来维护这个文档。初次可见的评论数量是受到限制的(对于这个例子来说，其受限数量是 3)；如果需要为特定帖子显示更多的评论，就需要查询第二个集合。
- 换句话说，它是一种用于快速加载用户墙页面的概要视图。

第四个集合是 social.posts，它用于快速呈现社交新闻界面。这就是显示所有帖子的界面。

就像第三个集合一样，第四个集合也是一个依赖集合。它包含与 user.wall 信息相同的大多数信息，因此为简洁起见，这个文档已经被简写为：

```
{
_id: ObjectId(...),
user_id: "user id",
postmonth: '2015_07',
postlists: [ ... ]
}
```

10.2.2 操作

这些模式是为了读取性能而优化的。

1. 浏览帖子

由于 social.posts 和 user.wall 集合是为了呈现新闻源或第二个集合的片段中的用户墙帖子而优化的，所以其查询非常简单。

这两个集合具有类似的模式，所以可以用相同的代码来支持其抓取操作。下面是用于相同抓取操作的伪代码。该函数使用了如下内容作为参数：

- 需要被查询的集合。
- 其数据需要被浏览的用户。
- 月份是一个可选参数；如果指定了月份，则应该列出日期小于或等于指定月份的所有帖子。

```
Function Fetch_Post_Details
(Parameters: CollectionName, View_User_ID, Month)
SET QueryDocument to {"User_id": View_User_ID}
```

```
    IF Month IS NOT NULL
    APPEND Month Filter ["Month":{"$lte":Month}] to QueryDocument
    Set O_Cursor = (resultset of the collection after applying the
QueryDocument filter)
    Set Cur = (sort O_Cursor by "month" in reverse order)
    while records are present in Cur
                    Print record
    End while
    End Function
```

上述函数会按照发生时间反序检索指定用户墙的所有帖子或新闻源。

在呈现帖子时，需要进行某些检查。以下是其中一些。

首先，当用户正在浏览其页面时，在呈现墙上的帖子时需要检查这些帖子是否可以在其自己的墙上显示。一个用户墙会包含该用户已经发布的或者其正在关注的用户的帖子。下面这个函数使用了两个参数：用户墙归属的用户以及要呈现的帖子：

```
    functionCheck_VisibleOnOwnWall
    (Parameters: user, post)
    While Loop_User IN user.Circles List
                    If post by = Loop_User
    return true
                    else
    return false
    end while
    end function
```

上面的循环会遍历在 user.profile 集合中指定的圈子，并且如果所涉及的帖子是由列表上的一个用户发布的，那么将返回 true。

除此之外，你还需要处理被阻止访问用户列表中的用户：

```
    function ReturnBlockedOrNot(user, post)
           if post by user id not in user blocked list
                    return true
           else
                    return false
    endfunction
```

在用户浏览另一个用户的墙时，你还需要处理权限检查：

```
    Function visibleposts(parameter user, post)
    if post circles is public
                    return true
    If post circles is public to all followed users
           Return true
    setlistofcircles = followers circle whose user_id is the post's by id.
```

```
if listofcircles in post's circles
      return true
return false

end function
```

这个函数首先会检查该帖子的圈子是否是公开。如果是公开，则该帖子将被显示给所有用户。

如果该帖子的圈子没有被设置为公开，那么它将被显示给关注该用户的用户。如果这些条件都不满足，那么将转向关注当前登录用户的所有用户的圈子。如果该圈子的列表位于帖子圈子的列表中，则表明用户位于接受该帖子的圈子中，因此该帖子将是可见的。如果这些条件都不满足，那么该帖子将不会对该用户显示。

为了得到更好的性能，social.posts 和 user.wall 这两个集合中的 user_id 和 month 字段上都需要一个索引。

2. 创建评论

为了在包含指定文本的指定帖子上创建来自一个用户的评论，需要执行类似于如下所示的代码：

```
Function postcomment(
Parameters: commentedby, commentedonpostid, commenttext)
Set commentedon to current datetime
Set month to month of commentedon
Set comment document as {"by": {id: commentedby[id], "Name":
commentedby["name"]},
"ts": commentedon, "text": commenttext}
Update user.posts collection. Push comment document.
Update user.walls collection. Push the comment document.
Increment the comments_shown in user.walls collection by 1.
Update social.posts collection. Push the comment document.
Increment the comments_shown counter in social.posts collection by 1.
End function
```

由于你正在显示的评论数在这两个依赖集合(user.wall 和 social.posts 集合)中最多为 3 条，因此需要定期运行以下更新语句：

```
Function MaintainComments
SET MaximumComments = 3
Loop through social.posts
        If posts.comments_shown>MaximumComments
                Pop the comment which was inserted first
                Decrement comments_shown by 1
        End if
Loop through user.wall
```

```
        If posts.comments_shown>MaximumComments
                Pop the comment which was inserted first
                Decrement comments_shown by 1
        End if

    End loop
    End Function
```

要快速执行这些更新，需要在 posts.id 和 posts.comments_shown 上创建索引。

创建新帖子

这段代码中的基本操作顺序如下：

(1) 帖子首先会被保存到“记录的系统”中，也就是 user.posts 集合中。

(2) 接下来，会用这个帖子更新 user.wall 集合。

(3) 最后，会用这个帖子更新处于该帖子圈中的每个人的 social.posts 集合。

```
    Function createnewpost
    (parameter createdby, posttype, postdetail, circles)
    Set ts = current timestamp.
    Set month = month of ts
    Set post_document = {"ts": ts, "by":{id:createdby[id], name:
createdby[name]},
    "circles":circles, "type":posttype, "details":postdetails}
    Insert post_document into users.post collection
    Append post_document into user.walls collection
    Set userlist = all users who's circled in the post based on the posts circle
and the posted user id
    While users in userlist
    Append post_document to userssocial.posts collection
    End while
    End function
```

10.2.3 分片

可以通过对上面提及的 4 个集合分片来实现扩展性。由于 user.profile、user.wall 和 social.posts 包含了特定于用户的文档，因此对于这些集合来说 user_id 是一个完美的分片键。_id 是用于 users.post 集合的最佳分片键。

10.3 本章小结

在本章中，你使用了两个用例来了解如何才能将 MongoDB 用于解决特定问题。在下一章中，我们将列出 MongoDB 的限制以及它不适用的用例。

第 11 章

MongoDB 使用限制

“在开始使用一个新的数据库时，你还应该了解其使用限制，以便能够更好地使用它。”

在本章中，我们会列出 MongoDB 的使用限制，以及它不适用的用例。

11.1 MongoDB 的空间过大(对于 MMAPv1 而言)

我们首先探讨硬盘空间的问题。MongoDB(使用 MMAPv1 存储引擎)的空间过大了；也就是说，数据目录文件比数据库的实际数据要大。

这是由于预先分配数据文件造成的。根据设计，这是为了避免文件系统碎片。

数据目录中文件的名称形如<dbname>.0、<dbname>.1，以此类推。由 mongod 分配的第一个文件的大小是 64MB；所有后续的文件大小都会增加两倍，因此第二个文件大小为 128MB，第三个文件大小为 256MB，以此类推直到到达 2GB，之后的所有文件大小都将是 2GB。尽管此空间是在创建时就分配给数据文件的，但可能有些文件的空间有 90%都没有用到。对于较大的数据库来说，这一未使用的已分配空间大多数很小。

- 可以通过使用--noprealloc 选项来禁用该设置。不过，不建议将其用在生产环境中，它应该仅被用于测试以及具有小数据集的地方，其中会频繁进行删除数据库的操作。
- Oplog：如果 mongod 是一个副本集成员，那么数据目录中就会有一个名称为 oplog.rs 的文件。这个文件存在于本地数据库中，并且是一个预先分配的固定集合。在 64 位的安装程序中，用于这个文件的配置默认大约为硬盘空间的 5%。
- **日志**：日志文件也包含在数据目录中，这个数据目录会在 MongoDB 将硬盘上的写操作应用到数据库之前存储这些写操作。
- MongoDB 为日志预先分配了 3GB 的数据空间，这要大于实际的数据库大小，从而让它不适用于小型安装配置。为此，可行的变通方法是在你的命令行标记或 /etc/mongod.conf 文件中使用–smallflags，直到你的运行环境具有所需的硬盘空间为止。但此功能会让它不适用于小型安装配置。
- **空记录**：当文档或集合被删除时，其空间永远不会被操作系统回收；相反，MongoDB 会维护一份这些空记录的清单，它们可以被重复使用。

要回收这一被删除的空间，可以使用 compact 或 repairDatabase 选项，但要注意，这两个选项都需要额外的硬盘空间来运行。

提示： WiredTiger 存储引擎中不存在这样的限制。相反，其存储大小会减少到 50%，因为压缩了数据文件。还有，一旦集合被删除，那么硬盘空间就会被自动回收，这不同于上面提及的 MMAPv1 存储引擎。

11.2 内存问题(对于 MMAPv1 而言)

在 MongoDB 中，内存是由映射整个数据集的内存来管理的。它允许 OS 控制内存映射并且分配最大数量的 RAM。其结果是，性能并非最优，并且可以推断其内存使用并非高效。

1. 索引占用内存较多；换句话说，索引会消耗大量 RAM。由于这些索引都是 B 树索引，因此定义许多索引会导致系统资源的更快速消耗。

2. 这种情形的后果就是，在需要时会自动分配内存。在分片环境中，运行数据库会比较棘手。一般而言，就像所有数据库服务器一样，最好在一台专用的服务器上运行 MongoDB。

11.3 32 位与 64 位对比

MongoDB 有两个版本，32 位和 64 位。

由于 MongoDB 使用了内存映射的文件，因此 32 位版本被限制为只能存储大约 2GB 的数据。如果需要存储更多的数据，则应该使用 64 位版本。

从版本 3.0 开始，MongoDB 不再提供对 32 位版本的商业支持了。另外，MongoDB 的 32 位版本也不支持 WiredTiger 存储引擎。

11.4 BSON 文档

本节会介绍 BSON 文档的使用限制。

- **大小限制：** 就像使用其他数据库一样，文档中可以存储哪些内容是有限制的。当前版本支持最大 16MB 的文档。这一最大大小会确保一个文档无法使用过多的 RAM 或者在传输时使用过多的带宽。
- **嵌套深度限制：** 在 MongoDB 中，BSON 文档不支持超过 100 的嵌套层级。
- **字段名称：** 如果使用键“col1”存储 1000 个文档，那么这个键就会在数据集中被存储同样多次。尽管 MongoDB 支持任何文档，但实际上大多数字段名称都是相同的。保持简短的字段名称被视作优化空间使用的好办法。

11.5 命名空间使用限制

要注意命名空间方面的以下使用限制。

- **一个命名空间的长度**：包括集合与数据库在内的每一个命名空间，其长度都必须小于 123 个字节。
- **命名空间文件大小**(对于 MMAPv1 存储引擎而言)：一个命名空间文件的大小不能超过 2047MB。其默认大小为 16MB；不过，可以使用 nssize 选项来配置它。
- **命名空间的数量**(对于 MMAPv1 存储引擎而言)：命名空间的数量=(命名空间文件大小/628)。一个 16MB 的命名空间文件将支持大约 24 000 个命名空间。

提示：对于 WiredTiger 存储引擎来说不存在这样的限制。

11.6 索引使用限制

本节将介绍 MongoDB 中索引的使用限制。

- **索引大小**：索引条目不能大于 1024 字节。
- **每个集合的索引数量**：每个集合最多允许使用 64 个索引。
- **索引名称长度**：默认情况下，索引名称由字段名称和索引方向构成。包括命名空间(也就是数据库和集合名称)在内的索引名称不能超过 128 个字节。
  如果默认的索引名称变得过长，则可以将一个索引名称显式指定到 ensureIndex() 帮助器。
- **分片集合中的唯一索引**：只有当完整分片键作为一个前缀包含在唯一索引中时，它才支持跨分片。否则，唯一索引就无法支持跨分片。在这种情况下，会跨完整键而不是单个字段强制实现唯一性。
- **复合索引中的索引字段数量**：其数量不能超过 31 个字段。

11.7 固定集合使用限制——固定集合中文档的最大数量

如果将 max 参数用于指定一个固定集合中文档的最大数量，那么该集合中的文档就不能超过 232 个。不过，如果未使用这样的参数，那么文档数量就不受限制。

11.8 分片使用限制

分片是跨分片划分数据的机制。后面几节将探讨处理分片时需要清楚的一些使用限制。

11.8.1 及早分片以避免出现问题

使用分片键，数据就会被划分成数据块，然后它们会在分片之间自动分布。不过，如果延后实现分片，则可能会造成服务器的运行速度降低，因为数据块的划分和迁移需要时间和资源。

一个简单的解决方案是，使用像 MongoDB 云管理器这样的工具来监控 MongoDB 的实例容量(刷新时间、锁百分比、队列长度以及错误都是很好的指标)，并且在达到估计容量的 80%之前进行分片。

11.8.2 不能更新分片键

一旦文档被插入到集合中，就不能更新分片键了，因为 MongoDB 使用分片键来判定应该将文档路由到哪个分片。如果希望变更一个文档的分片键，那么推荐的解决方案是，移除该文档并且在完成变更之后重新插入该文档。

11.8.3 分片集合使用限制

应该在集合达到 256GB 之前对其进行分片。

11.8.4 选择合适的分片键

选择一个合适的分片键非常重要，因为一旦选择了分片键，那么要调整它就不那么容易了。

提示：什么样的分片键被视为不合适的分片键，完全取决于应用程序。假设该应用程序是一个新闻源；选择一个时间戳字段作为分片键就会是一个不合适的分片键，因为这最终只会从一个分片上插入、查询和迁移数据，而不是从整个群集这样做。如果需要调整分片键，通常会使用的处理过程就是转储并恢复集合。

11.9 安全性限制

对于数据库来说，安全性是一个重要的因素。我们从安全性方面来看看 MongoDB 的限制。

11.9.1 默认情况下没有身份验证

尽管默认情况下身份验证没有启用，但它是完全被支持的，并且可以被轻易启用。

11.9.2 与 MongoDB 的交互通信没有被加密

默认情况下，与 MongoDB 交互的连接是没有被加密的。当运行在公共网络上时，应该考虑对通信进行加密；否则你的数据安全性将受到威胁。可以使用支持 SSL 的 MongoDB 版本来加密公共网络上的通信，该功能仅在 64 位版本中可用。

11.10 写入和读取限制

后面几节将介绍重要的限制内容。

11.10.1 大小写敏感的查询

默认情况下 MongoDB 是对大小写敏感的。

例如，后面两个命令将返回不同的结果：db.books.find({name: 'PracticalMongoDB'}) 和 db.books.find({name: 'practicalmongodb'})。你应该确保你清楚所存储的数据是大写还是小写。尽管可以使用像 db.books.find({name: /practicalmongodb/i})这样的正则表达式搜索，但它们并非理想的搜索，因为它们相对很慢。

11.10.2 类型敏感的字段

由于 MongoDB 中没有强制文档的模式，因此它无法知道你正在犯错。你必须确保将正确的类型用于数据。

11.10.3 没有联结

MongoDB 中不支持联结。如果需要从多个集合中检索数据，则必须进行多次查询。不过，可以重新设计模式来将相关的数据放在一起，以便可以在单个查询中检索信息。

11.10.4 事务

MongoDB 仅支持单文档原子性。由于一个写操作可能会修改多个文档，因此此操作并非原子的。不过，可以使用隔离操作符来隔离影响多个文档的写操作。

副本集限制——副本集成员的数量

副本集被用于确保 MongoDB 中的数据冗余。一个成员充当主成员，而其余成员充当辅助成员。由于 MongoDB 使用的投票方式，你必须使用奇数个成员。

这是因为一个节点需要多数票才能成为主节点。如果使用偶数个节点，则最终会出现没有选中主节点的平局状态，因为没有一个成员会具有多数票。在这种情况下，该副

本集将变成只读。

可以使用仲裁者来打破这样的平局。它们可以帮助支持故障转移并且节省成本。要学习更多与副本集功能有关的内容，请参阅第 7 章。

11.11 MongoDB 不适用的范围

MongoDB 不适用于以下方面：

- 像会计或银行系统这样的高事务性系统。传统的 RDBMS 仍旧更加适合于这样的应用程序，它们要求大量的原子性复杂事务。
- 传统的商业智能应用程序，其中一个特定问题的 BI 数据库会生成高度优化的查询。对于这样的应用程序来说，数据仓库可能是一个更加适合的选择。
- 需要复杂 SQL 查询的应用程序。
- MongoDB 不支持事务操作，因此银行系统肯定不能使用它。

11.12 本章小结

在本章中，学习了与 MongoDB 的使用限制以及它不适用的用例有关的内容。

在下一章中，将介绍如何使用 MongoDB。

第 12 章

MongoDB 的最佳实践

“开始使用 MongoDB 很容易，但一旦你开始开发应用程序，将碰到可能需要最佳实践来实现特定用例的情况。”

在前面几章中，你逐步熟悉了 MongoDB。本章的目的是借助其他用户的经验概述已知的问题，但也提供了各种实践做法，这些实践做法有助于你学习使用 MongoDB 的过程变得容易一些。

如你所知，MongoDB 会用到文档，将 RAM 用于存储数据来提升性能，并且使用复制和分片来进一步提供数据安全性和可扩展性。

本章将介绍你应该清楚的一些技巧，从部署策略以强化查询到监控数据的安全性和一致性。

12.1 部署

在确定部署策略时，要牢记以下技巧以便硬件配置合适。这些技巧也会帮助你决定是否使用分片和复制。

- **数据集大小：**最重要的事情是确定当前的以及预期的数据集大小。这不仅会让你可以选择用于单独物理节点的资源，还有助于规划你的分片计划(如果需要的话)。
- **数据重要性：**第二重要的事情是确定数据的重要性，以确定数据的重要程度以及你能承受多大的数据丢失或数据延迟(尤其是在复制时)。
- **内存大小：**下一步是识别出需要的内存并且相应地处理 RAM。
  就像其他面向数据的应用程序一样，当整个数据集可以驻留在内存中时，MongoDB 的运行性能也会最佳，因而可以避免任何类型的硬盘 I/O。
  页面错误表明可能超出了可用的部署内存，并且你应该考虑增加内存。页面错误是一个指标，可以使用像 MongoDB 云管理器这样的监控工具来测量它。
  如果可能的话，你应该总是选择一个内存大小大于工作集大小的平台。
  如果工作集大小超过了单个节点的内存大小，那么你就应该考虑使用分片，这样一来就可以增加可用的内存数量。这样就会最大化总体部署的性能。

- **硬盘类型**：如果速度不是一个主要的关注点，或者如果数据集大于所有内存策略可以支持的大小，那么选择一种合适的硬盘类型就非常重要。IOPS(Input/Output Operations Per Second，每秒的输入/输出操作)是选择一种硬盘类型的关键；IOPS 越高，MongoDB 的性能就越好。如果可能的话，应该使用本地硬盘，因为网络存储会造成低性能以及高延迟的情况。还建议在创建磁盘阵列时使用 RAID 10。
- **CPU**：如果预期要使用映射还原，那么时钟频率以及可用的处理器就会变成重要的考虑事项。当运行一个大部分数据位于内存中的 mongod 时，时钟频率也会对总体性能产生重大影响。在你希望最大化每秒操作数量的情况下，你必须考虑在你的部署策略中包含具有高时钟/总线速度的 CPU。
- 如果高可用性是其中一个需求，则要使用**复制**。在任何 MongoDB 部署中，设置一个至少具有 3 个节点的副本集都应该成为标准。

  对于具有 3 个节点的复制来说，2x1 的部署是最常见的配置，其中有两个节点位于一个数据中心，另外一个备份节点位于次级数据中心，如图 12-1 所描述的。

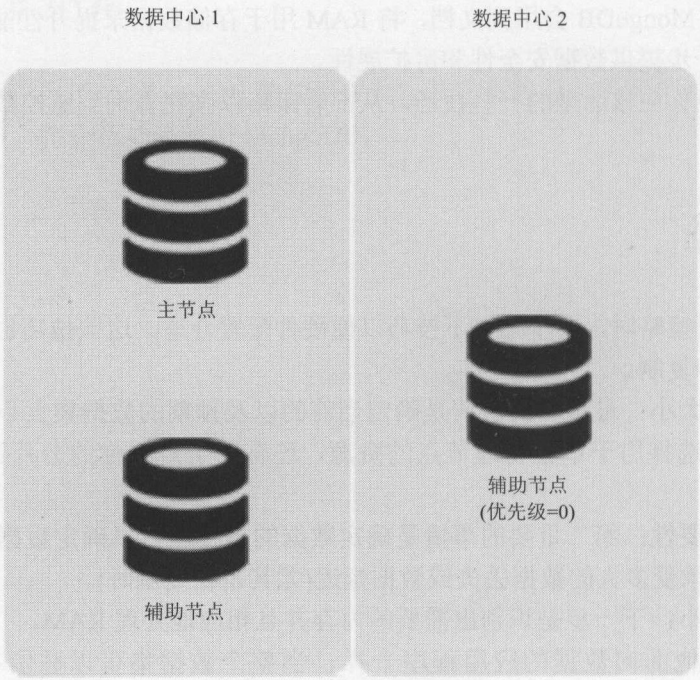

图 12-1　MongoDB 2x1 部署

12.1.1　MongoDB 网站的硬件配置建议

以下内容仅用于为 MongoDB 部署提供高级别的指导。实际的硬件配置取决于你的数据、可用性需求、查询、性能指标以及所选择的硬件组件的性能。

- **内存**：由于 MongoDB 为得到更好的性能而广泛使用了内存，因此越多的内存就意味着越好的性能。
- **存储**：MongoDB 可以使用 SSD(Solid State Drives，固态硬盘)或本地附加存储。由于 MongoDB 的硬盘访问模式不具有序列化特性，因此 SSD 的使用可以让客户体验到巨大的性能提升。使用 SSD 的另一个好处是，如果工作集不再适合放在内存中，那么它们提供了性能的平稳降级。

  大多数 MongoDB 部署都应该使用 RAID-10。

  在使用 WiredTiger 存储引擎时，出于性能问题的考虑，强烈建议使用 XFS 文件系统。

  另外，不要使用大型页面，因为 MongoDB 在使用默认虚拟内存页面时会执行得更好。
- **CPU**：由于使用 MMAPv1 存储引擎的 MongoDB 很少会碰到需要大量 CPU 内核的工作负荷，因此最好使用具有较快时钟频率的服务器，而不是使用具有多个内核但时钟频率较慢的服务器。不过，WiredTiger 存储引擎会密集使用 CPU，所以使用一台具有多个内核的服务器将提供显著的性能改善。

12.1.2 要注意的一些要点

作为本节的概要阐述，在为 MongoDB 选择硬件时，需要考虑以下重要的要点：

(1) 较快的 CPU 时钟频率和更多 RAM 对于生产效率很重要。

(2) 由于 MongoDB 不执行大量的计算，因此在使用 MMAPv1 存储引擎时，增加内核的数量会有所帮助，但不会带来很大的边际回报。

(3) 使用 SATA SSD 和 PCI(Peripheral Component Interconnect，外围组件互联)会提供良好的性价比以及很好的收益。

(4) 大量采购 SATA 硬盘将带来更高的效率。

(5) NUMA(Non-Uniform Memory Access，非一致性内存访问架构)硬件上的 MongoDB：这部分内容仅适用于运行在 Linux 上的 mongod，而非运行在 Windows 或其他类 Unix 系统上的实例。NUMA 和 MongoDB 并不能无缝地集成运行，因此在 NUMA 硬件上运行 MongoDB 时，需要为 MongoDB 禁用 NUMA，并且要使用交错存储策略来运行，因为 NUMA 会对 MongoDB 造成大量的操作性问题，其中包括一段时间的性能迟缓或高处理器使用。

12.2 编码

一旦获得了硬件，那么在使用该数据库进行编码时就可以考虑以下技巧：

- 第一个要点是思考要用于满足指定应用程序需求的数据模型，并且决定是使用嵌入还是引用，还是两者混合使用。需要在较快的性能与实时一致性的保障之间做出平衡，因此基于你的应用程序才能决定。

- 要避免会导致文档大小无限增长的应用程序模式。在 MongoDB 中，BSON 文档的最大大小是 16MB。应该避免让文档以无限方式增长的应用程序模式。

例如，应用程序不应该用导致文档显著增长的方式来更新文档。当文档大小超出所分配的大小时，MongoDB 会迁移该文档。此过程不仅消耗时间，还会占用大量资源并且会不必要地减缓其他数据库操作。此外，它还会导致存储的低效使用。

注意，上面提及的限制仅适用于 MMAPv1 存储引擎。在使用 WiredTiger 时，每次更新都会重写文档。

例如，我们来思考一个博客应用程序。在这个应用程序中，要估算博文帖子会收到多少个回应是很难的。该应用程序被设置为仅对用户显示评论的一个子集，比如最近的评论或者前 10 条评论。在这种情况下，相较于创建一个博文帖子和用户回应被维护成单个文档的嵌入模型，你应该创建一个引用模型，其中每一个回应或回应分组都会被维护成单独的文档，然后在这些文档中添加一个到该博文帖子的引用。这样一来，就可以控制文档的无限增长，而如果遵循嵌入数据的第一个模型，就会出现这种情况。

- 你也可以设计未来使用的文档。尽管 MongoDB 提供了在需要时在文档中附加新字段的选项，但它有一个缺点。当引入新字段时，可能会出现当前可用空间不足以满足文档使用的情况，从而导致 MongoDB 为该文档寻找一个新的空间并且将它移动到这个新空间中，而这会需要时间。因此，如果清楚结构的话，那么一开始就创建好所有的字段总是最好的方式，无论当时是否有可用的数据。正如上面所重点强调的，空间将被分配给该文档，且无论何时有值，仅需要更新即可。这样一来，MongoDB 就不必寻找空间了；它只需要更新输入的值即可，这一过程会快得多。
- 在合适的地方，你也可以使用预期的大小来创建文档。这一点也是为了确保将足够的空间分配给文档，并且未来任何增长都不会导致手忙脚乱地导出寻找空间。这可以通过使用一个垃圾回收字段来实现，该字段包含一个初始插入文档时使用的预期大小的字符串，然后会立即释放该字段：

```
>mydbcol.insert({"_id" : ObjectID(..),......, "tempField" :
stringOfAnticipatedSize})
>mydbcol.update({"_id" : ...}, {"$unset" : {"tempField" : 1}})
```

- 当知道并且总是知道你所访问的字段名称时，就应该总是使用子文档。否则，就要使用数组。
- 如果希望查询必须被计算并且不会显式呈现在文档中的信息，那么最佳的选择是在文档中明确该信息。因为 MongoDB 旨在仅存储和检索数据，所以它不会进行计算。所有琐碎的计算都会被推到客户端，从而导致性能问题。
- 另外，要尽可能避免使用$Where，因为它是一个非常耗时和消耗资源的操作。

- 在设计文档时要使用正确的数据类型。例如，一个数字仅应该被存储为一个数字数据类型，而不是一个字符串数据类型。使用字符串需要更多的空间来存储数据，并且会对在该数据上执行的操作产生影响。
- 要注意的另一件事是，MongoDB 中的字符串是区分大小写的。因此搜索“practicalMongoDB”将不会找到“Practicalmongodb”。

  因此，在进行字符串搜索时，可以进行以下操作之一：
  - 以规范的大小写格式存储数据。
  - 在搜索时使用/I 这个正则表达式。
  - 在聚合框架中使用$toUpper 或$toLower。
- 使用你自己的唯一键作为_id 将节省一些空间，并且当计划对该键索引时会很有用。不过，在决定将你自己的键用作_id 时，需要牢记以下几点：
  - 必须确保键的唯一性。
  - 另外，要考虑键的插入顺序，因为插入顺序将确定要使用多少 RAM 来维护这个索引。
- 按需检索字段。当每秒钟要满足数百或数千个请求时，仅抓取所需的字段肯定是有好处的。
- 仅将 GridFS 用作存储大于单个文档所提供大小的数据，或者用作存储过大而无法在客户端上一次加载的数据，例如视频。可以用流的方式返回给客户端的任何内容都适用于 GridFS。
- 使用 TTL 来删除文档。如果一个集合中的文档在一段预先定义的时间段之后需要被删除，则可以使用 TTL 功能在到达预先定义的时间之后自动删除该文档。

假设你有一个集合，它维护包含用户和系统交互详情的文档。该文档有一个名称为 lastActivity 的数据字段，它会跟踪用户的系统的交互。我们假设你有一个需求，需要仅存留用户会话一小时。在这种情况下，可以为字段 lastActivity 将 TTL 设置为 3600 秒。一个后台线程将自动运行并且将检查和删除闲置超过 3600 秒的文档。

- 如果需要基于插入顺序的高吞吐量，则可以使用固定集合。在有些场景中，根据数据大小的不同，需要在系统中维护一个滚动的数据窗口。例如，可以使用一个固定集合来存储高容量系统的日志信息，以便快速检索最近的日志条目。
- 注意，如果不仔细的话，MongoDB 的灵活模式可能会导致出现不一致的数据。例如，在未正确更新的情况下复制数据(嵌入文档)的能力会导致数据不一致等。因此检查数据一致性非常重要。
- 尽管 MongoDB 会应对无缝的故障转移，按照良好的编程习惯，应该精心编写应用程序来处理任何异常以及优雅地应对这样的情况。

12.3 应用程序响应时间优化

一旦你开始开发应用程序，其中一个最重要的需求就是设置一个可以接受的响应时间。换句话说，应用程序应该及时响应。可以将以下技巧用于优化：

- 尽可能地避免硬盘访问和页面错误。积极弄明白应用程序未来所需的数据集大小，以便处理和添加更多的内存，从而避免页面错误和硬盘读取。另外，以主要访问内存中可用数据的方式来编写你的应用程序，这样页面错误就不会频繁出现。
- 在要查询的字段上创建一个索引。如果在执行的筛选条件上创建索引，那么索引在内存中存储这一方式就会导致较少的内存消耗，因而也就会对查询产生积极影响。
- 如果应用程序涉及返回一些字段而非完整文档结构的查询，则要创建覆盖索引。
- 使用会被最多查询用到的复合索引也会节省内存，因为相较于在内存中加载多个索引，一个索引就足够了。
- 要在正则表达式中使用结束通配符，以便从相关索引中获得益处。
- 要尝试创建可彻底减少从中进行选择的可能文档数量的索引。字段“Gender”上的索引所带来的好处就小于字段“Phone Number”上索引能带来的好处。
- 索引并非总是好的。需要维持所用索引的最佳平衡。尽管你应该创建支持你的查询的索引，但你还是应该记得删除不再被使用的索引，因为每一个索引都具有与插入/更新操作有关的资源开销。如果有一个索引未被使用却仍然存在，则它会对整体的数据库能力产生负面影响。这对于以插入操作为主的工作负荷来说尤为重要。
- 应该以分层方式设计文档，其中相关的内容会被分组到一起，并且在合适的位置被描述为层次结构。这会使 MongoDB 找出期望的信息，而无须扫描整个文档。
- 在应用 AND 操作符时，你总是应该先查询小的结果集，再查询较大的结果集，因为这样就能查询少量的文档。如果清楚最严格的条件，则应该首先处理该条件。
- 在使用 OR 进行查询时，你应该从较大的结果集移动到较小的结果集，因为这会限制用于后续查询的搜索空间。
- 工作集应该适合内存大小。
- 将 WiredTiger 存储引擎用于以写入为主且 I/O 密集型的应用程序，因为它支持数据块以及索引级别的压缩。

12.4 数据安全性

你学习了在决定你的部署时需要牢记哪些要点；还学习了得到良好性能的一些重要技巧。现在我们来看看用于数据安全性和一致性方面的一些技巧：

- 复制和日志是为数据安全性提供的两种方法。通常建议使用复制运行生产环境而非使用单台服务器运行它。并且你至少应该日志化其中一台服务器。当无法启用复制并且运行在单台服务器上时，日志就能提供数据安全保障。第 8 章阐释了启用日志时写操作是如何进行的。
- 在一台服务器崩溃的情况下，修复应该是恢复数据的最后手段。尽管在运行修复之后，数据库可能不会被损坏，但它将不会包含所有的数据。
- 在复制环境中，要将 W 设置为多数安全写入。这是为了确保写操作被复制到副本集的多数成员。尽管这会降低写操作的速度，但写操作会是安全的。
- 在运行该命令时应该总是将 wtimeout 与 w 一起指定，以避免无限的等待时间。
- MongoDB 应该总是运行在具有规则的一个受信环境中，以阻止来自所有未知系统、机器或网络的访问。

12.5 管理

下面是一些管理技巧：

- 使用持久服务器的瞬时备份。要使用启用日志的数据库的备份，可以使用一个文件系统快照或者进行一次普通的文件同步+锁定，然后转储。注意，你不能只复制所有的文件，而不进行文件同步和锁定，因为复制不是一个瞬时操作。
- 应该使用修复来压缩数据库，因为它实际上会进行一次 mongodump，然后进行一次 mongorestore，从而得到数据的一个干净副本，并且在该过程中，会移除你数据文件中的所有空隙“空间”。
- MongoDB 提供了数据库分析工具。它会记录所有数据库操作的细粒度信息。可以启用它来记录所有事件的信息或者仅记录超出可配置阈值期间的事件，该阈值默认为 100ms。

提示：该分析工具数据存储在一个固定集合中。相较于解析日志文件而言，查询这个集合可能更为容易。

- 可以使用一个阐释计划来查看一个查询是如何被处理的。这涉及一些相关信息，比如使用了哪个索引、返回了多少个文档、该索引是否覆盖了这个查询、扫描了多少个索引条目，以及该查询使用了多少以毫秒计算的时间来返回结果。当一个查询的处理时间小于 1ms 时，该阐释计划会显示 0。当调用该阐释计划时，它会丢弃旧的计划并且初始化验证可用索引的过程，以确保使用最佳的可行计划。

12.6 复制延迟

复制延迟是监控副本集背后的主要管理关注点。一个指定辅助节点的复制延迟，其差异在于一个操作被写入主节点的时间以及相同操作被复制到该辅助节点的时间。通常，

复制延迟会自我纠正并且是短暂的。不过，如果其延迟时间仍旧很长并且持续发生，那么系统可能就出现了问题。你可能最终要关闭系统直到问题被解决，或者可能需要人工介入来调整不匹配的问题，或者你甚至可能最终需要运行带有过期数据的系统。

可以使用以下命令来判定副本集的当前复制延迟：

```
testset:PRIMARY>rs.printSlaveReplicationInfo()
```

此外，可以使用 rs.printReplicationInfo()命令来填充缺失的数据片段：

```
testset:PRIMARY>rs.printReplicationInfo()
```

也可以使用 MongoDB 云管理器来查看最近以及历史的复制延迟信息。可以从每个辅助节点的 Status 选项卡上使用该复制延迟图形。

这里有一些帮助减少延迟时间的技巧：

- 在有大量写入负荷的场景中，你应该使用一个像主节点一样性能强劲的辅助节点，以便它能保持与主节点同步，并且写操作可以相同速度被应用到这个辅助节点。另外，你应该使用足够的网络带宽，以便可以操作被创建的相同速度从主节点检索这些操作。
- 调整应用程序的写关注。
- 如果辅助节点被用于索引构建，那么可以规划在主节点上只有很少的写入操作时来进行。
- 如果辅助节点被用于进行备份，则可以考虑在不阻塞的情况下进行备份。
- 检查复制错误。运行 rs.status()并且检查 errmsg 字段。此外，可以检查辅助节点的日志文件来查看所有存在的错误消息。

12.7 分片

当数据不再适合放在一个节点上时，就可以使用分片来确保数据跨群集平均分布，并且操作不会受到资源限制的影响。

- 选择一个合适的分片键。
- 必须在生产环境部署中使用 3 台配置服务器以便提供冗余。
- 在集合达到 256GB 之前就对集合进行分片。

12.8 监控

应该主动监控 MongoDB 系统以检测不寻常的行为，从而能够采取必要的操作来解决问题。有几个工具可用于监控 MongoDB 的部署。

MongoDB 开发者提供了一个名称为 MongoDB 云管理器的免费托管的监控服务。MongoDB 云管理器提供了整个群集指标的仪表盘视图。或者，可以使用 nagios、SNMP

或 munin 来构建你自己的工具。

MongoDB 还提供了几个工具，比如 mongostat 和 mongotop，用于获取性能信息。在使用监控服务时，应该仔细观察以下内容：

- **操作计数器**：包括插入、删除、读取、更新和指针使用。
- **常驻内存**：应该总是关注所分配的内存。这个计数器值应该总是低于物理内存。如果内存消耗完了，则会体验到性能减缓，因为会出现页面错误和索引缺失。
- **工作集大小**：活动的工作集应该适合内存大小才能得到良好的性能，因此需要密切注意工作集。可以优化查询，以便工作集能够适合内存大小，也可以在工作集预期会增长的情况下增加内存大小。
- **队列**：在 MongoDB 3.0 版本之前，读取者-写入者锁被用于写入者锁使用的同步读取和独占访问。在这样的情况下，你可能最终要在单个写入者背后使用队列，它可能会包含读取/写入查询。需要将队列指标与锁百分比一起监控，如果队列和锁百分比的趋势是上升的，则表明你的数据库中存在竞争。将操作修改为批量模式或者修改数据模型会对并发性产生显著、积极的影响。从版本 3.0 开始，引入了集合级别的锁(在 MMAPv1 存储引擎中)以及文档级别的锁(在 WiredTiger 存储引擎中)。这就使得并发性得到了提升，因为数据库级别不再需要任何独占式访问的写入锁了。因此从这个版本开始，你只需要测量队列指标即可。
- 无论何时应用程序中出现不顺畅的情况，CRUD 行为、索引模式以及索引都可以帮助你更好地理解应用程序的流程。
- 建议针对完全大小的数据库运行整体的性能测试，比如生产环境的数据库副本，因为在处理实际的数据时，性能特性往往是被重点关注的。这也使得可以避免在处理实际的性能数据库时碰到可能突然出现的意外情况。

12.9 本章小结

在本章中，我们介绍了各种最佳实践以帮助你熟悉了解 MongoDB。

FOR PROFESSIONALS BY PROFESSIONALS®

MongoDB实战

《MongoDB实战 架构、开发与管理》首先简要阐述NoSQL数据库的基础知识，然后介绍了MongoDB——业界领先的基于文档的NoSQL数据库，让读者逐步了解MongoDB方方面面的内容。

本书涵盖了数据模型、底层架构、使用Mongo Shell编码、管理MongoDB平台以及其他主题。本书还提供了使用MongoDB平台进行架构、开发与部署应用程序的清晰指导与实践示例。数据库开发人员、架构师和管理员将在本书中找到涵盖MongoDB平台所有知识点的有用信息，以及如何将它用于实践。

最近几年，由于各种各样NoSQL数据库的涌现，就传统RDBMS而言的“一刀切”的想法受到了挑战。如今市场上有超过120种NoSQL数据库可用，并且目前处于领先地位的就是MongoDB。随着如此众多的公司选择MongoDB作为其NoSQL数据库选项，如何结合专业建议以便最大化利用该软件的实践需求也就越来越大了。

本书主要内容：

- 深刻理解NoSQL数据库
- 理解如何开始使用MongoDB
- 系统讲解MongoDB的架构、开发和管理
- 众多的“如何实践”让你可以最为有效地使用该技术来解决你面临的问题

清华大学出版社数字出版网站
WQBook 文泉局
www.wqbook.com

Apress®
www.apress.com

ISBN 978-7-302-45673-5
9 787302 456735 >
定价：49.80元